

Java DB Reference Manual

Version 10.5

Derby Document build:
August 10, 2009, 9:51:45 AM (EDT)

Contents

Apache Software FoundationJava DB Reference ManualApache Derby

Copyright



Related information

License

License

The Apache License, Version 2.0

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use,
      reproduction, and distribution as defined by Sections 1 through
      9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized
      by the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under
      common control with that entity. For the purposes of this
      definition, "control" means (i) the power, direct or indirect,
      to cause the direction or management of such entity, whether by
      contract or otherwise, or (ii) ownership of fifty percent (50%)
      or more of the outstanding shares, or (iii) beneficial ownership
      of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making
      modifications, including but not limited to software source code,
      documentation source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or
      Object form, that is based on (or derived from) the Work and
      for which the editorial revisions, annotations, elaborations,
      or other modifications represent, as a whole, an original work
      of authorship.  For the purposes of this License, Derivative
      Works shall not include works that remain separable from, or
      merely link (or bind by name) to the interfaces of, the Work
      and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or
      additions to that Work or Derivative Works thereof, that is
      intentionally submitted to Licensor for inclusion in the Work
      by the copyright owner or by an individual or Legal Entity
      authorized to submit on behalf of the copyright owner. For the
      purposes of this definition,
      "submitted" means any form of electronic, verbal, or written
      communication sent to the Licensor or its representatives,
      including but not limited to communication on electronic mailing
      lists, source code control systems, and issue tracking systems
```

```
      that are managed by, or on behalf of, the Licensor for the
      purpose of discussing and improving the Work, but excluding
      communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a
      Contribution."

      "Contributor" shall mean Licensor and any individual or Legal
      Entity on behalf of whom a Contribution has been received by
      Licensor and subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions
      of this License, each Contributor hereby grants to You a
      perpetual, worldwide, non-exclusive, no-charge, royalty-free,
      irrevocable copyright license to reproduce, prepare Derivative
      Works of, publicly display, publicly perform, sublicense, and
      distribute the Work and such Derivative Works in Source or
      Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have
      made, use, offer to sell, sell, import, and otherwise transfer
      the Work, where such license applies only to those patent claims
      licensable by such Contributor that are necessarily infringed by
      their Contribution(s) alone or by combination of their
      Contribution(s) with the Work to which such Contribution(s) was
      submitted. If You institute patent litigation against any entity
      (including a cross-claim or counterclaim in a lawsuit) alleging
      that the Work or a Contribution incorporated within the Work
      constitutes direct or contributory patent infringement, then any
      patent licenses granted to You under this License for that Work
      shall terminate as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of its
          distribution, then any Derivative Works that You distribute
          must include a readable copy of the attribution notices
          contained within such NOTICE file, excluding those notices
          that do not pertain to any part of the Derivative Works, in
          at least one of the following places: within a NOTICE text
          file distributed as part of the Derivative Works; within the
          Source form or documentation, if provided along with the
          Derivative Works; or, within a display generated by the
          Derivative Works, if and wherever such third-party notices
          normally appear. The contents of the NOTICE file are for
          informational purposes only and do not modify the License.
          You may add Your own attribution notices within Derivative
          Works that You distribute, alongside or as an addendum to
          the NOTICE text from the Work, provided that such additional
          attribution notices cannot be construed as modifying the
          License.

      You may add Your own copyright statement to Your modifications
```

```
      and may provide additional or different license terms and
      conditions for use, reproduction, or distribution of Your
      modifications, or for any such Derivative Works as a whole,
      provided Your use, reproduction, and distribution of the Work
      otherwise complies with the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state
      otherwise, any Contribution intentionally submitted for
      inclusion in the Work by You to the Licensor shall be under the
      terms and conditions of this License, without any additional
      terms or conditions.  Notwithstanding the above, nothing herein
      shall supersede or modify the terms of any separate license
      agreement you may have executed with Licensor regarding such
      Contributions.

   6. Trademarks. This License does not grant permission to use the
      trade names, trademarks, service marks, or product names of the
      Licensor, except as required for reasonable and customary use
      in describing the origin of the Work and reproducing the content
      of the NOTICE file.

   7. Disclaimer of Warranty. Unless required by applicable law or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or
      conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or
      FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for
      determining the appropriateness of using or redistributing the
      Work and assume any risks associated with Your exercise of
      permissions under this License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and
      grossly negligent acts) or agreed to in writing, shall any
      Contributor be liable to You for damages, including any direct,
      indirect, special, incidental, or consequential damages of any
      character arising as a result of this License or out of the use
      or inability to use the Work (including but not limited to
      damages for loss of goodwill, work stoppage, computer failure or
      malfunction, or any and all other commercial damages or losses),
      even if such Contributor has been advised of the possibility of
      such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by
      reason of your accepting any such warranty or additional
      liability.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the following
      boilerplate notice, with the fields enclosed by brackets "[]"
      replaced with your own identifying information. (Don't include
      the brackets!)  The text should be enclosed in the appropriate
      comment syntax for the file format. We also recommend that a
      file or class name and description of purpose be included on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.
```

Relationship between Java(TM) DB and Derby

Java(TM) DB is a relational database management system that is based on the Java programming language and SQL. Java DB is a commercial release of the Apache Software Foundation's (ASF) open source relational database project. The Apache project is called Derby.

The Java DB product includes Derby without any modification whatsoever to the underlying source code.

Technical support is available for purchase for the Java DB product through Sun.

Because Java DB and Derby have the same functionality, the Java DB documentation refers to the core functionality as Derby.

Java DB Version 10.5 is based on the Derby Version 10.5 release. References to "Derby" in the Java DB documentation refer to the Version 10.5 release of Apache Derby.

About this guide

For general information about the Derby documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with Java DB*.

Purpose of this document

This book, the *Java DB Reference Manual*, provides reference information about Derby. It covers Derby's SQL language, the Derby implementation of JDBC, Derby system catalogs, Derby error messages, Derby properties, and SQL keywords.

Audience

This book is a reference for Derby users, typically application developers. Derby users who are not familiar with the SQL standard or the Java programming language will benefit from consulting books on those topics.

Derby users who want a how-to approach to working with Derby or an introduction to Derby concepts should read the *Java DB Developer's Guide*.

How this guide is organized

This guide includes the following sections:

- *SQL language reference*

  Reference information about Derby's SQL language, including manual pages for statements, functions, and other syntax elements.
- *SQL reserved words*

  SQL keywords beyond the standard SQL-92 keywords.
- *Derby support for SQL-92 features*

  A list of SQL-92 features that Derby does and does not support.
- *Derby system tables*

  Reference information about the Derby system catalogs.
- *Derby exception messages and SQL states*

  Information about Derby exception messages.
- *JDBC reference*

  Information about Derby's implementation of the Java Database Connectivity (JDBC) API.
- *Setting attributes for the database connection URL*

  Information about the supported attributes to Derby's JDBC database connection URL.
- *Derby property reference*

  Information about Derby properties.
- *J2EE Compliance: Java Transaction API and javax.sql Interfaces*

  Information about Derby's support for the Java EE platform, in particular support for the Java Transaction API and the JDBC API.
- *Derby API*

  Notes about proprietary APIs for Derby.
- *Supported territories*

Territories supported by Derby.

- *Derby limitations*

  Limitations of Derby.

SQL language reference

Derby implements an SQL-92 core subset, as well as some SQL-99 features.

This section provides an overview of the current SQL language by describing the statements, built-in functions, data types, expressions, and special characters it contains.

Capitalization and special characters

Using the classes and methods of JDBC, you submit SQL statements to Derby as strings. The character set permitted for strings containing SQL statements is Unicode. Within these strings, the following rules apply:

- Double quotation marks delimit special identifiers referred to in SQL-92 as *delimited identifiers*.
- Single quotation marks delimit character strings.
- Within a character string, to represent a single quotation mark or apostrophe, use two single quotation marks. (In other words, a single quotation mark is the escape character for a single quotation mark.)

  A double quotation mark does not need an escape character. To represent a double quotation mark, simply use a double quotation mark. However, note that in a Java program, a double quotation mark requires the backslash escape character.

  Example:

  ```
  -- a single quotation mark is the escape character
  -- for a single quotation mark

  VALUES 'Joe''s umbrella'
  -- in ij, you don't need to escape the double quotation marks
  VALUES 'He said, "hello!"'

  n = stmt.executeUpdate(
      "UPDATE aTable setStringcol = 'He said, \"hello!\"'");
  ```
- SQL keywords are case-insensitive. For example, you can type the keyword SELECT as SELECT, Select, select, or sELECT.
- SQL-92-style identifiers are case-insensitive (see *SQL92Identifier*), unless they are delimited.
- Java-style identifiers are always case-sensitive.
- * is a wildcard within a *SelectExpression.* See The * wildcard. It can also be the multiplication operator. In all other cases, it is a syntactical metasymbol that flags items you can repeat 0 or more times.
- % and _ are character wildcards when used within character strings following a LIKE operator (except when escaped with an escape character). See Boolean expressions.
- Comments can be either single- or multiline as per the SQL-92 standard. Singleline comments start with two dashes (--) and end with the newline character. Multiline comments are bracketed and start with forward slash star (/*), and end with star forward slash (*/). Note that bracketed comments may be nested. Any text between the starting and ending comment character sequence is ignored.

SQL identifiers

An *identifier* is the representation within the language of items created by the user, as opposed to language keywords or commands. Some identifiers stand for *dictionary objects*, which are the objects you create- such as tables, views, indexes, columns, and

constraints- that are stored in a database. They are called dictionary objects because Derby stores information about them in the system tables, sometimes known as a data dictionary. SQL also defines ways to alias these objects within certain statements.

Each kind of identifier must conform to a different set of rules. Identifiers representing dictionary objects must conform to SQL-92 identifier rules and are thus called *SQL92Identifiers*.

Rules for SQL92 identifiers

Ordinary identifiers are identifiers not surrounded by double quotation marks. Delimited identifiers are identifiers surrounded by double quotation marks.

An ordinary identifier must begin with a letter and contain only letters, underscore characters (_), and digits. The permitted letters and digits include all Unicode letters and digits, but Derby does not attempt to ensure that the characters in identifiers are valid in the database's locale.

A delimited identifier can contain any characters within the double quotation marks. The enclosing double quotation marks are not part of the identifier; they serve only to mark its beginning and end. Spaces at the end of a delimited identifier are insignificant (truncated). Derby translates two consecutive double quotation marks within a delimited identifier as one double quotation mark-that is, the "translated" double quotation mark becomes a character in the delimited identifier.

Periods within delimited identifiers are not separators but are part of the identifier (the name of the dictionary object being represented).

So, in the following example:

```
"A.B"
```

is a dictionary object, while

```
"A"."B"
```

is a dictionary object qualified by another dictionary object (such as a column named "B" within the table "A").

SQL92Identifier

An *SQL92Identifier* is a dictionary object identifier that conforms to the rules of SQL-92. SQL-92 states that identifiers for dictionary objects are limited to 128 characters and are case-insensitive (unless delimited by double quotes), because they are automatically translated into uppercase by the system. You cannot use reserved words as identifiers for dictionary objects unless they are delimited. If you attempt to use a name longer than 128 characters, *SQLException* X0X11 is raised.

Derby defines keywords beyond those specified by the SQL-92 standard (see SQL reserved words).

Example

```
-- the view name is stored in the
-- system catalogs as ANIDENTIFIER
CREATE VIEW AnIdentifier (RECEIVED) AS VALUES 1
-- the view name is stored in the system
-- catalogs with case intact
CREATE VIEW "ACaseSensitiveIdentifier" (RECEIVED) AS VALUES 1
```

This section describes the rules for using *SQL92Identifiers* to represent the following dictionary objects.

Qualifying dictionary objects

Since some dictionary objects can be contained within other objects, you can qualify those dictionary object names. Each component is separated from the next by a period. An *SQL92Identifier* is "dot-separated." You qualify a dictionary object name in order to avoid ambiguity.

column-Name

In many places in the SQL syntax, you can represent the name of a column by qualifying it with a *table-Name* or *correlation-Name*.

In some situations, you cannot qualify a *column-Name* with a *table-Name* or a *correlation-Name,* but must use a *Simple-column-Name* instead. Those situations are:

- creating a table (CREATE TABLE statement)
- specifying updatable columns in a cursor
- in a column's correlation name in a SELECT expression (see *SelectExpression*)
- in a column's correlation name in a *TableExpression* (see *TableExpression*)

You cannot use correlation-Names for updatable columns; using correlation-Names in this way will cause an SQL exception. For example:

```
SELECT c11 AS col1, c12 AS col2, c13 FROM t1 FOR UPDATE of c11,c13
```

In this example, the correlation-Name `col1 FOR c11` is not permitted because `c11` is listed in the FOR UPDATE list of columns. You can use the correlation-Name `FOR c12` because it is not in the FOR UPDATE list.

Syntax

```
[ { table-Name | correlation-Name } . ] SQL92Identifier
```

Example

```
-- C.Country is a column-Name qualified with a
-- correlation-Name.
SELECT C.Country
FROM APP.Countries C
```

correlation-Name

A *correlation-Name* is given to a table expression in a FROM clause as a new name or alias for that table. You do not qualify a *correlation-Name* with a *schema-Name*.

You cannot use correlation-Names for updatable columns; using correlation-Names in this way will cause an SQL exception. For example:

```
SELECT c11 AS col1, c12 AS col2, c13 FROM t1 FOR UPDATE of c11,c13
```

In this example, the correlation-Name `col1 FOR c11` is not permitted because `c11` is listed in the FOR UPDATE list of columns. You can use the correlation-Name `FOR c12` because it is not in the FOR UPDATE list.

Syntax

```
SQL92Identifier
```

Example

```
-- C is a correlation-Name
SELECT C.NAME
FROM SAMP.STAFF C
```

new-table-Name

A *new-table-Name* represents a renamed table. You cannot qualify a *new-table-Name* with a *schema-Name*.

Syntax

```
SQL92Identifier
```

Example

```
-- FlightBooks is a new-table-Name that does not include a schema-Name
RENAME TABLE FLIGHTAVAILABILITY TO FLIGHTAVAILABLE
```

schemaName

A *schemaName* represents a *schema*. Schemas contain other dictionary objects, such as tables and indexes. Schemas provide a way to name a subset of tables and other dictionary objects within a database.

You can explicitly create or drop a schema. The default user schema is the *APP* schema (if no user name is specified at connection time). You cannot create objects in schemas starting with SYS.

Thus, you can qualify references to tables with the schema name. When a schemaName is not specified, the default schema name is implicitly inserted. System tables are placed in the *SYS* schema. You must qualify all references to system tables with the *SYS* schema identifier. For more information about system tables, see Derby system tables.

A schema is hierarchically the highest level of dictionary object, so you cannot qualify a schemaName.

Syntax

```
SQL92Identifier
```

Example

```
-- SAMP.EMPLOYEE is a table-Name qualified by a schemaName
SELECT COUNT(*) FROM SAMP.EMPLOYEE
-- You must qualify system catalog names with their schema, SYS
SELECT COUNT(*) FROM SYS.SysColumns
```

Simple-column-Name

A *Simple-column-Name* is used to represent a column when it cannot be qualified by a *table-Name* or *correlation-Name*. This is the case when the qualification is fixed, as it is in a column definition within a CREATE TABLE statement.

Syntax

```
SQL92Identifier
```

Example

```
-- country is a Simple-column-Name
CREATE TABLE CONTINENT (COUNTRY VARCHAR(26) NOT NULL PRIMARY KEY,
COUNTRY_ISO_CODE CHAR(2), REGION VARCHAR(26))
```

synonym-Name

A *synonym-Name* represents a synonym for a table or a view. You can qualify a *synonym-Name* with a *schema-Name*.

Syntax

```
[ schemaName. ] SQL92Identifier
```

table-Name

A *table-Name* represents a table. You can qualify a *table-Name* with a *schemaName*.

Syntax

```
[ schemaName. ] SQL92Identifier
```

Example

```
-- SAMP.PROJECT is a table-Name that includes a schemaName
SELECT COUNT(*) FROM SAMP.PROJECT
```

view-Name

A *view-Name* represents a table or a view. You can qualify a *view-Name* with a *schema-Name*.

Syntax

```
[ schemaName. ] SQL92Identifier
```

Example

```
-- This is a View qualified by a schema-Name
SELECT COUNT(*) FROM SAMP.EMP_RESUME
```

index-Name

An *index-Name* represents an index. Indexes live in schemas, so you can qualify their names with *schema-Names*. Indexes on system tables are in the *SYS* schema.

Syntax

```
[ schemaName . ] SQL92Identifier
```

Example

```
DROP INDEX APP.ORIGINDEX;
-- OrigIndex is an index-Name without a schema-Name
CREATE INDEX ORIGINDEX ON FLIGHTS (ORIG_AIRPORT)
```

constraint-Name

You cannot qualify constraint-names.

Syntax

```
SQL92Identifier
```

Example

```
-- country_fk2 is a constraint name
CREATE TABLE DETAILED_MAPS (COUNTRY_ISO_CODE CHAR(2)
CONSTRAINT country_fk2 REFERENCES COUNTRIES)
```

cursor-Name

A *cursor-Name* refers to a cursor. No SQL language command exists to *assign* a name to a cursor. Instead, you use the JDBC API to assign names to cursors or to retrieve system-generated names. For more information, see the *Java DB Developer's Guide*. If you assign a name to a cursor, you can refer to that name from within SQL statements.

You cannot qualify a *cursor-Name*.

Syntax

```
SQL92Identifier
```

Example

```
stmt.executeUpdate("UPDATE SAMP.STAFF SET COMM = " +
"COMM + 20 " + "WHERE CURRENT OF " + ResultSet.getCursorName());
```

TriggerName

A *TriggerName* refers to a trigger created by a user.

Syntax

```
[ schemaName . ] SQL92Identifier
```

Example

```
DROP TRIGGER TRIG1
```

AuthorizationIdentifier

User names within the Derby system are known as *authorization identifiers*. The authorization identifier represents the name of the user, if one has been provided in the connection request. The default schema for a user is equal to its authorization identifier. User names can be case-sensitive within the authentication system, but they are always case-insensitive within Derby's authorization system unless they are delimited. For more information, see the *Java DB Developer's Guide*.

Syntax

```
SQL92Identifier
```

Example

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'Amber,FRED')
```

RoleName

A *RoleName* refers to an SQL role. A role in a database is uniquely identified by its role name.

Syntax

```
SQL92Identifier
```

In terms of SQL, a role name is also technically an *AuthorizationIdentifier*, but that term is often used for user names in Derby for historical reasons.

Example

```
DROP ROLE reader
```

Statements

This section provides manual pages for both high-level language constructs and parts thereof. For example, the CREATE INDEX statement is a high-level statement that you can execute directly via the JDBC interface. This section also includes clauses, which are not high-level statements and which you cannot execute directly but only as part of a high-level statement. The ORDER BY and WHERE clauses are examples of this kind of clause. Finally, this section also includes some syntactically complex portions of statements called expressions, for example *SelectExpression* and *TableSubquery.* These clauses and expressions receive their own manual pages for ease of reference.

Unless it is explicitly stated otherwise, you can execute or prepare and then execute all the high-level statements, which are all marked with the word *statement*, via the interfaces provided by JDBC. This manual indicates whether an expression can be executed as a high-level statement.

The sections provide general information about statement use, and descriptions of the specific statements.

Interaction with the dependency system

Derby internally tracks the dependencies of prepared statements, which are SQL statements that are precompiled before being executed. Typically they are prepared (precompiled) once and executed multiple times.

Prepared statements depend on the dictionary objects and statements they reference. (Dictionary objects include tables, columns, constraints, indexes, views, and triggers.) Removing or modifying the dictionary objects or statements on which they depend invalidates them internally, which means that Derby will automatically try to recompile the statement when you execute it. If the statement fails to recompile, the execution request fails. However, if you take some action to restore the broken dependency (such as restoring the missing table), you can execute the same prepared statement, because Derby will recompile it automatically at the next execute request.

Statements depend on one another-an UPDATE WHERE CURRENT statement depends on the statement it references. Removing the statement on which it depends invalidates the UPDATE WHERE CURRENT statement.

In addition, prepared statements prevent execution of certain DDL statements if there are open results sets on them.

Manual pages for each statement detail what actions would invalidate that statement, if prepared.

Here is an example using the Derby tool ij:

```
ij> CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
ij> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
-- this example uses the ij command prepare,
-- which prepares a statement
ij> prepare p1 AS 'INSERT INTO MyTable VALUES (4)';
-- p1 depends on mytable;
ij> execute p1;
1 row inserted/updated/deleted
-- Derby  executes it without recompiling
```

```
ij> CREATE INDEX i1 ON mytable(mycol);
0 rows inserted/updated/deleted
-- p1 is temporarily invalidated because of new index
ij> execute p1;
1 row inserted/updated/deleted
-- Derby  automatically recompiles p1 and executes it
ij> DROP TABLE mytable;
0 rows inserted/updated/deleted
-- Derby  permits you to drop table
-- because result set of p1 is closed
-- however, the statement p1 is temporarily invalidated
ij> CREATE TABLE mytable (mycol INT);
0 rows inserted/updated/deleted
ij> INSERT INTO mytable VALUES (1), (2), (3);
3 rows inserted/updated/deleted
ij> execute p1;
1 row inserted/updated/deleted
-- Because p1 is invalid, Derby  tries to recompile it
-- before executing.
-- It is successful and executes.
ij> DROP TABLE mytable;
0 rows inserted/updated/deleted
-- statement p1 is now invalid,
-- and this time the attempt to recompile it
-- upon execution will fail
ij> execute p1;
ERROR 42X05: Table/View 'MYTABLE' does not exist.
```

ALTER TABLE statement

The ALTER TABLE statement allows you to:

- add a column to a table
- add a constraint to a table
- drop a column from a table
- drop an existing constraint from a table
- increase the width of a VARCHAR or VARCHAR FOR BIT DATA column
- override row-level locking for the table (or drop the override)
- change the increment value and start value of the identity column
- change the nullability constraint for a column
- change the default value for a column

Syntax

```
ALTER TABLE table-Name
{
    ADD COLUMN column-definition |
    ADD CONSTRAINT clause |
    DROP [ COLUMN ] column-name [ CASCADE | RESTRICT ]
    DROP { PRIMARY KEY | FOREIGN KEY constraint-name | UNIQUE
  constraint-name | CHECK constraint-name | CONSTRAINT constraint-name }
    ALTER [ COLUMN ] column-alteration |
    LOCKSIZE { ROW | TABLE }
}
```

column-definition

```
Simple-column-Name [ DataType ]
[ Column-level-constraint ]*
[ [ WITH ] DEFAULT DefaultConstantExpression
  | generation-clause
]
```

The syntax for the *column-definition* for a new column is a subset of the syntax for a column in a CREATE TABLE statement.

The syntax of *DataType* is described in Data types. The *DataType* can be omitted only if you specify a *generation-clause*. If you omit the *DataType*, the type of the generated column is the type of the *generation-clause*. If you specify both a *DataType* and a *generation-clause*, the type of the *generation-clause* must be assignable to *DataType*.

For details on *DefaultConstantExpression*, see Column default.

column-alteration

```
column-Name SET DATA TYPE VARCHAR(integer) |
column-Name SET DATA TYPE VARCHAR FOR
 BIT DATA(integer)
 |
column-name SET INCREMENT BY integer-constant |
column-name RESTART WITH integer-constant |
column-name [ NOT ] NULL
 |
column-name [ WITH | SET ] DEFAULT default-value |
column-name DROP DEFAULT
```

In the column-alteration, SET INCREMENT BY integer-constant, specifies the interval between consecutive values of the identity column. The next value to be generated for the identity column will be determined from the last assigned value with the increment applied. The column must already be defined with the IDENTITY attribute.

RESTART WITH integer-constant specifies the next value to be generated for the identity column. RESTART WITH is useful for a table that has an identity column that was defined as GENERATED BY DEFAULT and that has a unique key defined on that identity column. Because GENERATED BY DEFAULT allows both manual inserts and system generated values, it is possible that manually inserted values can conflict with system generated values. To work around such conflicts, use the RESTART WITH syntax to specify the next value that will be generated for the identity column. Consider the following example, which involves a combination of automatically generated data and manually inserted data:

```
CREATE TABLE tauto(i INT GENERATED BY DEFAULT AS IDENTITY, k INT)
CREATE UNIQUE INDEX tautoInd ON tauto(i)
INSERT INTO tauto(k) values 1,2
```

The system will automatically generate values for the identity column. But now you need to manually insert some data into the identity column:

```
INSERT INTO tauto VALUES (3,3)
INSERT INTO tauto VALUES (4,4)
INSERT INTO tauto VALUES (5,5)
```

The identity column has used values 1 through 5 at this point. If you now want the system to generate a value, the system will generate a 3, which will result in a unique key exception because the value 3 has already been manually inserted. To compensate for the manual inserts, issue an ALTER TABLE statement for the identity column with RESTART WITH 6:

```
ALTER TABLE tauto ALTER COLUMN i RESTART WITH 6
```

ALTER TABLE does not affect any view that references the table being altered. This includes views that have an "*" in their SELECT list. You must drop and re-create those views if you wish them to return the new columns.

Derby raises an error if you try to change the *DataType* of a generated column to a type which is not assignable from the type of the *generation-clause*. Derby also raises an error if you try to add a DEFAULT clause to a generated column.

Adding columns

The syntax for the *column-definition* for a new column is almost the same as for a column in a CREATE TABLE statement. This syntax allows a column constraint to be placed on the new column within the ALTER TABLE ADD COLUMN statement. However, a column with a NOT NULL constraint can be added to an existing table if you give a default value; otherwise, an exception is thrown when the ALTER TABLE statement is executed.

Just as in CREATE TABLE, if the column definition includes a primary key constraint, the column cannot contain null values, so the NOT NULL attribute must also be specified (SQLSTATE 42831).

Note: If a table has an UPDATE trigger without an explicit column list, adding a column to that table in effect adds that column to the implicit update column list upon which the trigger is defined, and all references to transition variables are invalidated so that they pick up the new column.

If you add a generated column to a table, Derby computes the generated values for all existing rows in the table.

Adding constraints

ALTER TABLE ADD CONSTRAINT adds a table-level constraint to an existing table. Any supported table-level constraint type can be added via ALTER TABLE. The following limitations exist on adding a constraint to an existing table:

- When adding a foreign key or check constraint to an existing table, Derby checks the table to make sure existing rows satisfy the constraint. If any row is invalid, Derby throws a statement exception and the constraint is not added.
- All columns included in a primary key must contain non null data and be unique.

  ALTER TABLE ADD UNIQUE or PRIMARY KEY provide a shorthand method of defining a primary key composed of a single column. If PRIMARY KEY is specified in the definition of column C, the effect is the same as if the PRIMARY KEY(C) clause were specified as a separate clause. The column cannot contain null values, so the NOT NULL attribute must also be specified.

For information on the syntax of constraints, see CONSTRAINT clause. Use the syntax for table-level constraint when adding a constraint with the ADD TABLE ADD CONSTRAINT syntax.

Dropping columns

ALTER TABLE DROP COLUMN allows you to drop a column from a table.

The keyword COLUMN is optional.

The keywords CASCADE and RESTRICT are also optional. If you specify neither CASCADE nor RESTRICT, the default is CASCADE.

If you specify RESTRICT, then the column drop will be rejected if it would cause a dependent schema object to become invalid.

If you specify CASCADE, then the column drop should additionally drop other schema objects which have become invalid.

The schema objects which can cause a DROP COLUMN RESTRICT to be rejected include: views, triggers, primary key constraints, foreign key constraints, unique key constraints, check constraints, and column privileges. If one of these types of objects depends on the column being dropped, DROP COLUMN RESTRICT will reject the statement.

Derby also raises an error if you specify RESTRICT when you drop a column referenced by the *generation-clause* of a generated column. However, if you specify CASCADE, the generated column is also dropped with CASCADE semantics.

You may not drop the last (only) column in a table.

CASCADE/RESTRICT doesn't consider whether the column being dropped is used in any indexes. When a column is dropped, it is removed from any indexes which contain it. If that column was the only column in the index, the entire index is dropped.

Dropping constraints

ALTER TABLE DROP CONSTRAINT drops a constraint on an existing table. To drop an unnamed constraint, you must specify the generated constraint name stored in *SYS.SYSCONSTRAINTS* as a delimited identifier.

Dropping a primary key, unique, or foreign key constraint drops the physical index that enforces the constraint (also known as a *backing index*).

Modifying columns

The *column-alteration* allows you to alter the named column in the following ways:

- Increasing the width of an existing VARCHAR or VARCHAR FOR BIT DATA column. CHARACTER VARYING or CHAR VARYING can be used as synonyms for the VARCHAR keyword.

  To increase the width of a column of these types, specify the data type and new size after the column name.

  You are not allowed to decrease the width or to change the data type. You are not allowed to increase the width of a column that is part of a primary or unique key referenced by a foreign key constraint or that is part of a foreign key constraint.
- Specifying the interval between consecutive values of the identity column.

  To set an interval between consecutive values of the identity column, specify the integer-constant. You must previously define the column with the IDENTITY attribute (SQLSTATE 42837). If there are existing rows in the table, the values in the column for which the SET INCREMENT default was added do not change.
- Modifying the nullability constraint of a column.

  You can add the NOT NULL constraint to an existing column. To do so there must not be existing NULL values for the column in the table.

  You can remove the NOT NULL constraint from an existing column. To do so the column must not be used in a PRIMARY KEY constraint.
- Changing the default value for a column.

  You can use DEFAULT default-value to change a column default. To disable a previously set default, use DROP DEFAULT (alternatively, you can specify NULL as the default-value).

Setting defaults

You can specify a default value for a new column. A default value is the value that is inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is NULL. If you add a default to a new column, existing rows in the table gain the default value in the new column.

For more information about defaults, see CREATE TABLE statement.

Changing the lock granularity for the table

The LOCKSIZE clause allows you to override row-level locking for the specific table, if your system uses the default setting of row-level locking. (If your system is set for table-level locking, you cannot change the locking granularity to row-level locking, although Derby allows you to use the LOCKSIZE clause in such a situation without throwing an exception.) To override row-level locking for the specific table, set locking for the table to TABLE. If you created the table with table-level locking granularity, you can change locking back to ROW with the LOCKSIZE clause in the ALTER TABLE STATEMENT. For information about why this is sometimes useful, see *Tuning Java DB*.

Examples

```
-- Add a new column with a column-level constraint
-- to an existing table
-- An exception will be thrown if the table
-- contains any rows
-- since the newcol will be initialized to NULL
-- in all existing rows in the table
ALTER TABLE CITIES ADD COLUMN REGION VARCHAR(26)
CONSTRAINT NEW_CONSTRAINT CHECK (REGION IS NOT NULL);

-- Add a new unique constraint to an existing table
-- An exception will be thrown if duplicate keys are found
ALTER TABLE SAMP.DEPARTMENT
ADD CONSTRAINT NEW_UNIQUE UNIQUE (DEPTNO);

-- add a new foreign key constraint to the
-- Cities table. Each row in Cities is checked
-- to make sure it satisfied the constraints.
-- if any rows don't satisfy the constraint, the
-- constraint is not added
ALTER TABLE CITIES ADD CONSTRAINT COUNTRY_FK
Foreign Key (COUNTRY) REFERENCES COUNTRIES (COUNTRY);

-- Add a primary key constraint to a table
-- First, create a new table
CREATE TABLE ACTIVITIES (CITY_ID INT NOT NULL,
SEASON CHAR(2), ACTIVITY VARCHAR(32) NOT NULL);
-- You will not be able to add this constraint if the
-- columns you are including in the primary key have
-- null data or duplicate values.
ALTER TABLE Activities ADD PRIMARY KEY (city_id, activity);

-- Drop the city_id column if there are no dependent objects:
ALTER TABLE Cities DROP COLUMN city_id RESTRICT;
-- Drop the city_id column, also dropping all dependent objects:
ALTER TABLE Cities DROP COLUMN city_id CASCADE;

-- Drop a primary key constraint from the CITIES table

ALTER TABLE Cities DROP CONSTRAINT Cities_PK;
-- Drop a foreign key constraint from the CITIES table
ALTER TABLE Cities DROP CONSTRAINT COUNTRIES_FK;
-- add a DEPTNO column with a default value of 1
ALTER TABLE SAMP.EMP_ACT ADD COLUMN DEPTNO INT DEFAULT 1;
-- increase the width of a VARCHAR column
ALTER TABLE SAMP.EMP_PHOTO ALTER PHOTO_FORMAT SET DATA TYPE VARCHAR(30);
-- change the lock granularity of a table
ALTER TABLE SAMP.SALES LOCKSIZE TABLE;

-- Remove the NOT NULL constraint from the MANAGER column
ALTER TABLE Employees ALTER COLUMN Manager NULL;
-- Add the NOT NULL constraint to the SSN column
ALTER TABLE Employees ALTER COLUMN ssn NOT NULL;

-- Change the default value for the SALARY column
ALTER TABLE Employees ALTER COLUMN Salary DEFAULT 1000.0
ALTER TABLE Employees ALTER COLUMN Salary DROP DEFAULT
```

Results

An ALTER TABLE statement causes all statements that are dependent on the table being altered to be recompiled before their next execution. ALTER TABLE is not allowed if there are any open cursors that reference the table being altered.

CALL (PROCEDURE) statement

The CALL (PROCEDURE) statement is used to call procedures. A call to a procedure does not return any value.

Syntax

```
CALL procedure-Name ( [ expression [, expression]* ] )
```

Example

```
CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,
    IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
    PARAMETER STYLE JAVA READS SQL DATA LANGUAGE JAVA EXTERNAL NAME
    'com.acme.sales.calculateRevenueByMonth';
CALL SALES.TOTAL_REVENUE(?,?,?);
```

CREATE statements

Use the CREATE statements to create functions, indexes, procedures, roles, schemas, synonyms, tables, triggers, and views.

CREATE FUNCTION statement

The CREATE FUNCTION statement allows you to create Java functions, which you can then use in an expression.

The function owner and the database owner automatically gain the EXECUTE privilege on the function, and are able to grant this privilege to other users. The EXECUTE privileges cannot be revoked from the function and database owners.

For details on how Derby matches functions to Java methods, see Argument matching.

Syntax

```
CREATE FUNCTION function-name ( [ FunctionParameter
   [, FunctionParameter] ] * ) RETURNS ReturnDataType [ FunctionElement ]
 *
```

function-Name

```
[ schemaName. ] SQL92Identifier
```

If schema-Name is not provided, the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with SYS.

FunctionParameter

```
[ parameter-Name ] DataType
```

PararameterName must be unique within a function.

The syntax of *DataType* is described in Data types.

Note: Data-types such as BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, and XML are not allowed as parameters in a CREATE FUNCTION statement.

ReturnDataType

```
TableType | DataType
```

The syntax of *DataType* is described in Data types.

TableType

```
TABLE( ColumnElement [, ColumnElement ]* )
```

This is the return type of a table function. Currently, only Derby-style table functions are supported. They are functions which return JDBC *ResultSet*s. For more information, see "Programming Derby-style table functions" in the *Java DB Developer's Guide*.

At run-time, as values are read out of the user-supplied *ResultSet*, Derby coerces those values to the data types declared in the CREATE FUNCTION statement. This affects values typed as CHAR, VARCHAR, LONG VARCHAR, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA, and DECIMAL/NUMERIC. Values which are too long are truncated to the maximum length declared in the CREATE FUNCTION statement. In addition, if a *String* value is returned in the *ResultSet* for a column of CHAR type and the *String* is shorter than the declared length of the CHAR column, Derby pads the end of the *String* with blanks in order to stretch it out to the declared length.

ColumnElement

```
SQL92IdentifierDataType
```

The syntax of *DataType* is described in Data types.

Note: XML is not allowed as the type of a column in the dataset returned by a table function.

FunctionElement

```
 {
| LANGUAGE { JAVA }
| DeterministicCharacteristic
| EXTERNAL NAME string
| PARAMETER STYLE ParameterStyle
| { NO SQL | CONTAINS SQL | READS SQL DATA }
| { RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT }
 }
```

LANGUAGE

JAVA- the database manager will call the function as a public static method in a Java class.

EXTERNAL NAME *string*
String describes the Java method to be called when the function is executed, and takes the following form:

```
class_name.method_name
```

The External Name cannot have any extraneous spaces.

DeterministicCharacteristic

```
DETERMINISTIC | NOT DETERMINISTIC
```

Declares that the function is deterministic, meaning that with the same set of input values, it always computes the same result. The default is NOT DETERMINISTIC. Derby cannot recognize whether an operation is actually deterministic, so you must take care to specify the `DeterministicCharacteristic` correctly.

ParameterStyle

```
JAVA | DERBY_JDBC_RESULT_SET
```

The function will use a parameter-passing convention that conforms to the Java language and SQL Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. Result sets can be returned through additional

parameters to the Java method of type java.sql.ResultSet[] that are passed single entry arrays.

Derby does not support long column types (for example Long Varchar, BLOB, and so on). An error will occur if you try to use one of these long column types.

The PARAMETER STYLE is DERBY_JDBC_RESULT_SET if and only if this is a Derby-style table function, that is, a function which returns TableType and which is mapped to a method which returns a JDBC *ResultSet*. Otherwise, the PARAMETER STYLE must be JAVA.

NO SQL, CONTAINS SQL, READS SQL DATA

Indicates whether the function issues any SQL statements and, if so, what type.

CONTAINS SQL
: Indicates that SQL statements that neither read nor modify SQL data can be executed by the function. Statements that are not supported in any function return a different error.

NO SQL
: Indicates that the function cannot execute any SQL statements

READS SQL DATA
: Indicates that some SQL statements that do not modify SQL data can be included in the function. Statements that are not supported in any stored function return a different error. This is the default value.

RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT
Specifies whether the function is called if any of the input arguments is null. The result is the null value.

RETURNS NULL ON NULL INPUT
: Specifies that the function is not invoked if any of the input arguments is null. The result is the null value.

CALLED ON NULL INPUT
: Specifies that the function is invoked if any or all input arguments are null. This specification means that the function must be coded to test for null argument values. The function can return a null or non-null value. This is the default setting.

The function elements may appear in any order, but each type of element can only appear once. A function definition must contain these elements:

- **LANGUAGE**
- **PARAMETER STYLE**
- **EXTERNAL NAME**

Example of declaring a scalar function

```
CREATE FUNCTION TO_DEGREES
( RADIANS DOUBLE )
RETURNS DOUBLE
PARAMETER STYLE JAVA
NO SQL LANGUAGE JAVA
EXTERNAL NAME 'java.lang.Math.toDegrees'
```

Example of declaring a table function

```
CREATE FUNCTION PROPERTY_FILE_READER
( FILENAME VARCHAR( 32672 ) )
RETURNS TABLE
  (
     KEY_COL     VARCHAR( 10 ),
     VALUE_COL VARCHAR( 1000 )
  )
```

```
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
NO SQL
EXTERNAL NAME 'vtis.example.PropertyFileVTI.propertyFileVTI'
```

CREATE INDEX statement

A CREATE INDEX statement creates an index on a table. Indexes can be on one or more columns in the table.

Syntax

```
CREATE [UNIQUE] INDEX index-Name
ON table-Name ( Simple-column-Name [ ASC | DESC ]
    [ , Simple-column-Name [ ASC | DESC ]] * )
```

The maximum number of columns for an index key in Derby is 16.

An index name cannot exceed 128 characters.

A column must not be named more than once in a single CREATE INDEX statement. Different indexes can name the same column, however.

Derby can use indexes to improve the performance of data manipulation statements (see *Tuning Java DB*). In addition, UNIQUE indexes provide a form of data integrity checking.

Index names are unique within a schema. (Some database systems allow different tables in a single schema to have indexes of the same name, but Derby does not.) Both index and table are assumed to be in the same schema if a schema name is specified for one of the names, but not the other. If schema names are specified for both index and table, an exception will be thrown if the schema names are not the same. If no schema name is specified for either table or index, the current schema is used.

By default, Derby uses the ascending order of each column to create the index. Specifying ASC after the column name does not alter the default behavior. The DESC keyword after the column name causes Derby to use descending order for the column to create the index. Using the descending order for a column can help improve the performance of queries that require the results in mixed sort order or descending order and for queries that select the minimum or maximum value of an indexed column.

If a qualified index name is specified, the schema name cannot begin with SYS.

Indexes and constraints

Unique, primary key, and foreign key constraints generate indexes that enforce or "back" the constraint (and are thus sometimes called *backing indexes*). If a column or set of columns has a UNIQUE or PRIMARY KEY constraint on it, you can not create an index on those columns. Derby has already created it for you with a system-generated name. System-generated names for indexes that back up constraints are easy to find by querying the system tables if you name your constraint. Adding a PRIMARY KEY or UNIQUE constraint when an existing UNIQUE index exists on the same set of columns will result in two physical indexes on the table for the same set of columns. One index is the original UNIQUE index and one is the backing index for the new constraint.

To find out the name of the index that backs a constraint called FLIGHTS_PK:

```
SELECT CONGLOMERATENAME FROM SYS.SYSCONGLOMERATES,
SYS.SYSCONSTRAINTS WHERE
SYS.SYSCONGLOMERATES.TABLEID = SYSCONSTRAINTS.TABLEID
AND CONSTRAINTNAME = 'FLIGHTS_PK'
```

```
CREATE INDEX OrigIndex ON Flights(orig_airport);
-- money is usually ordered from greatest to least,
-- so create the index using the descending order
CREATE INDEX PAY_DESC ON SAMP.EMPLOYEE (SALARY);
```

```
-- use a larger page size for the index
call
 SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.storage.pageSize','8192');
CREATE INDEX IXSALE ON SAMP.SALES (SALES);
call
 SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY('derby.storage.pageSize',NULL);
```

Page size and key size

Note: The size of the key columns in an index must be equal to or smaller than half the page size. If the length of the key columns in an existing row in a table is larger than half the page size of the index, creating an index on those key columns for the table will fail. This error only occurs when creating an index if an existing row in the table fails the criteria. After an index is created, inserts may fail if the size of their associated key exceeds the criteria.

Statement dependency system

Prepared statements that involve SELECT, INSERT, UPDATE, UPDATE WHERE CURRENT, DELETE, and DELETE WHERE CURRENT on the table referenced by the CREATE INDEX statement are invalidated when the index is created. Open cursors on the table are not affected.

CREATE PROCEDURE statement

The CREATE PROCEDURE statement allows you to create Java stored procedures, which you can then call using the CALL PROCEDURE statement.

The procedure owner and the database owner automatically gain the EXECUTE privilege on the procedure, and are able to grant this privilege to other users. The EXECUTE privileges cannot be revoked from the procedure and database owners.

For details on how Derby matches procedures to Java methods, see Argument matching.

Syntax

```
CREATE PROCEDURE procedure-Name ( [ ProcedureParameter
   [, ProcedureParameter] ] * )
[ ProcedureElement ] *
```

procedure-Name

```
[ schemaName. ] SQL92Identifier
```

If schema-Name is not provided, the current schema is the default schema. If a qualified procedure name is specified, the schema name cannot begin with SYS.

ProcedureParameter

```
[ { IN | OUT | INOUT } ] [ parameter-Name ] DataType
```

The default value for a parameter is IN. ParameterName must be unique within a procedure.

The syntax of *DataType* is described in Data types.

Note: Data-types such as BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, and XML are not allowed as parameters in a CREATE PROCEDURE statement.

ProcedureElement

```
 {
| [ DYNAMIC ] RESULT SETS INTEGER
| LANGUAGE { JAVA }
| DeterministicCharacteristic
| EXTERNAL NAME string
| PARAMETER STYLE JAVA
| { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
```

```
}
```

DYNAMIC RESULT SETS *integer*

Indicates the estimated upper bound of returned result sets for the procedure. Default is no (zero) dynamic result sets.

LANGUAGE

JAVA- the database manager will call the procedure as a public static method in a Java class.

EXTERNAL NAME *string*
String describes the Java method to be called when the procedure is executed, and takes the following form:

```
class_name.method_name
```

The External Name cannot have any extraneous spaces.

DeterministicCharacteristic

```
DETERMINISTIC | NOT DETERMINISTIC
```

Declares that the procedure is deterministic, meaning that with the same set of input values, it always computes the same result. The default is NOT DETERMINISTIC. Derby cannot recognize whether an operation is actually deterministic, so you must take care to specify the `DeterministicCharacteristic` correctly.

PARAMETER STYLE

JAVA - The procedure will use a parameter-passing convention that conforms to the Java language and SQL Routines specification. INOUT and OUT parameters will be passed as single entry arrays to facilitate returning values. Result sets are returned through additional parameters to the Java method of type java.sql.ResultSet [] that are passed single entry arrays.

Derby does not support long column types (for example Long Varchar, BLOB, and so on). An error will occur if you try to use one of these long column types.

NO SQL, CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA

Indicates whether the stored procedure issues any SQL statements and, if so, what type.

CONTAINS SQL
Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure. Statements that are not supported in any stored procedure return a different error. MODIFIES SQL DATA is the default value.

NO SQL
Indicates that the stored procedure cannot execute any SQL statements

READS SQL DATA
Indicates that some SQL statements that do not modify SQL data can be included in the stored procedure. Statements that are not supported in any stored procedure return a different error.

MODIFIES SQL DATA
Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures.

The procedure elements may appear in any order, but each type of element can only appear once. A procedure definition must contain these elements:

- **LANGUAGE**
- **PARAMETER STYLE**
- **EXTERNAL NAME**

Example

```
CREATE PROCEDURE SALES.TOTAL_REVENUE(IN S_MONTH INTEGER,
IN S_YEAR INTEGER, OUT TOTAL DECIMAL(10,2))
PARAMETER STYLE JAVA READS SQL DATA LANGUAGE JAVA EXTERNAL NAME
'com.acme.sales.calculateRevenueByMonth'
```

CREATE ROLE statement

The CREATE ROLE statement allows you to create an SQL role.

Only the database owner can create a role.

For more information on roles, see "Using SQL roles" in the *Java DB Developer's Guide*.

Syntax

```
CREATE ROLE roleName
```

Before you issue a CREATE ROLE statement, verify that the *derby.database.sqlAuthorization* property is set to TRUE. The *derby.database.sqlAuthorization* property enables SQL authorization mode.

You cannot create a role name if there is a user by that name. An attempt to create a role name that conflicts with an existing user name raises the *SQLException* X0Y68.

If user names are not controlled by the database owner (or administrator), it may be a good idea to use a naming convention for roles to reduce the possibility of collision with user names.

Derby tries to avoid name collision between user names and role names, but this is not always possible, because Derby has a pluggable authorization architecture. For example, an externally defined user may exist who has never yet connected to the database, created any schema objects, or been granted any privileges. If Derby knows about a user name, it will forbid creating a role with that name. Correspondingly, a user who has the same name as a role will not be allowed to connect. Derby built-in users are checked for collision when a role is created.

A role name cannot start with the prefix SYS (after case normalization). The purpose of this restriction is to reserve a name space for system-defined roles at a later point. Use of the prefix SYS raises the *SQLException* 4293A.

You cannot create a role with the name `PUBLIC` (after case normalization). `PUBLIC` is a reserved authorization identifier. An attempt to create a role with the name `PUBLIC` raises *SQLException* 4251B.

Example of creating a role

```
CREATE ROLE purchases_reader;
```

Examples of invalid role names

```
CREATE ROLE public;   -- throws SQLException;
CREATE ROLE "PUBLIC"; -- throws SQLException;
CREATE ROLE sysrole;  -- throws SQLException;
```

Example of creating a role using a naming convention

The following example uses the convention of giving every role name the suffix `_role`.

```
CREATE ROLE purchases_reader_role;
```

CREATE SCHEMA statement

A schema is a way to logically group objects in a single collection and provide a unique namespace for objects.

Syntax

```
CREATE SCHEMA { [ schemaName AUTHORIZATION user-name ] | [ schemaName ] |
[ AUTHORIZATION user-name ] }
```

The CREATE SCHEMA statement is used to create a schema. A schema name cannot exceed 128 characters. Schema names must be unique within the database.

The CREATE SCHEMA statement is subject to access control when the `derby.database.sqlAuthorization` property is set to `true` for the database or system. Only the database owner can create a schema with a name different from the current user name, and only the the database owner can specify

```
AUTHORIZATION user-name
```

with a user name other than the current user name. See "derby.database.sqlAuthorization" for information about the `derby.database.sqlAuthorization` property.

Note: Although the SQL standard allows you to specify any *AuthorizationIdentifier* as an AUTHORIZATION argument, Derby allows you to specify only a user, not a role.

CREATE SCHEMA examples

To create a schema for airline-related tables and give the authorization ID `anita` access to all of the objects that use the schema, use the following syntax:

```
CREATE SCHEMA FLIGHTS AUTHORIZATION anita
```

To create a schema employee-related tables, use the following syntax:

```
CREATE SCHEMA EMP
```

To create a schema that uses the same name as the authorization ID `takumi`, use the following syntax:

```
CREATE SCHEMA AUTHORIZATION takumi
```

To create a table called `availability` in the `EMP` and `FLIGHTS` schemas, use the following syntax:

```
CREATE TABLE FLIGHTS.AVAILABILITY
 (FLIGHT_ID CHAR(6) NOT NULL, SEGMENT_NUMBER INT NOT NULL,
 FLIGHT_DATE DATE NOT NULL, ECONOMY_SEATS_TAKEN INT,
 BUSINESS_SEATS_TAKEN INT, FIRSTCLASS_SEATS_TAKEN INT,
 CONSTRAINT FLT_AVAIL_PK
 PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER, FLIGHT_DATE))
```

```
CREATE TABLE EMP.AVAILABILITY
 (HOTEL_ID INT NOT NULL, BOOKING_DATE DATE NOT NULL, ROOMS_TAKEN INT,
 CONSTRAINT HOTELAVAIL_PK PRIMARY KEY (HOTEL_ID, BOOKING_DATE))
```

CREATE SYNONYM statement

Use the CREATE SYNONYM statement to provide an alternate name for a table or a view that is present in the same schema or another schema. You can also create synonyms for other synonyms, resulting in nested synonyms. A synonym can be used instead of the original qualified table or view name in SELECT, INSERT, UPDATE, DELETE or LOCK TABLE statements. You can create a synonym for a table or a view

that doesn't exist, but the target table or view must be present before the synonym can be used.

Synonyms share the same namespace as tables or views. You cannot create a synonym with the same name as a table that already exists in the same schema. Similarly, you cannot create a table or view with a name that matches a synonym already present.

A synonym can be defined for a table/view that does not exist when you create the synonym. If the table or view doesn't exist, you will receive a warning message (SQLSTATE 01522). The referenced object must be present when you use a synonym in a DML statement.

You can create a nested synonym (a synonym for another synonym), but any attempt to create a synonym that results in a circular reference will return an error message (SQLSTATE 42916).

Synonyms cannot be defined in system schemas. All schemas starting with 'SYS' are considered system schemas and are reserved by Derby.

A synonym cannot be defined on a temporary table. Attempting to define a synonym on a temporary table will return an error message (SQLSTATE XCL51).

Syntax

```
CREATE SYNONYM synonym-Name FOR { view-Name | table-Name }
```

The synonym-Name in the statement represents the synonym name you are giving the target table or view, while the view-Name or table-Name represents the original name of the target table or view.

Example

CREATE SYNONYM SAMP.T1 FOR SAMP.TABLEWITHLONGNAME

CREATE TABLE statement

A CREATE TABLE statement creates a table. Tables contain columns and constraints, rules to which data must conform. Table-level constraints specify a column or columns. Columns have a data type and can specify column constraints (column-level constraints).

The table owner and the database owner automatically gain the following privileges on the table and are able to grant these privileges to other users:

- INSERT
- SELECT
- REFERENCES
- TRIGGER
- UPDATE

These privileges cannot be revoked from the table and database owners.

For information about constraints, see CONSTRAINT clause.

You can specify a default value for a column. A default value is the value to be inserted into a column if no other value is specified. If not explicitly specified, the default value of a column is NULL. See Column default.

You can specify storage properties such as page size for a table by calling the `SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` system procedure.

If a qualified table name is specified, the schema name cannot begin with *SYS*.

Syntax

There are two different variants of the CREATE TABLE statement, depending on whether you are specifying the column definitions and constraints, or whether you are modeling the columns after the results of a query expression:

```
CREATE TABLE table-Name
  {
      ( {column-definition | Table-level constraint}
      [ , {column-definition | Table-level constraint} ] * )
  |
      [ ( column-name [ , column-name ] * ) ]
      AS query-expression
      WITH NO DATA
   }
```

Example

```
CREATE TABLE HOTELAVAILABILITY
     (HOTEL_ID INT NOT NULL, BOOKING_DATE DATE NOT NULL,
 ROOMS_TAKEN INT DEFAULT 0, PRIMARY KEY (HOTEL_ID, BOOKING_DATE));
-- the table-level primary key definition allows you to
-- include two columns in the primary key definition
PRIMARY KEY (hotel_id, booking_date))
-- assign an identity column attribute to an INTEGER
-- column, and also define a primary key constraint
-- on the column
CREATE TABLE PEOPLE
 (PERSON_ID INT NOT NULL GENERATED ALWAYS AS IDENTITY
 CONSTRAINT PEOPLE_PK PRIMARY KEY, PERSON VARCHAR(26));
-- assign an identity column attribute to a SMALLINT
-- column with an initial value of 5 and an increment value
-- of 5.
CREATE TABLE GROUPS
 (GROUP_ID SMALLINT NOT NULL GENERATED ALWAYS AS IDENTITY
 (START WITH 5, INCREMENT BY 5), ADDRESS VARCHAR(100), PHONE
 VARCHAR(15));
```

Note: For more examples of CREATE TABLE statements using the various constraints, see CONSTRAINT clause.

CREATE TABLE ... AS ...

With the alternate form of the CREATE TABLE statement, the column names and/or the column data types can be specified by providing a query. The columns in the query result are used as a model for creating the columns in the new table.

If no column names are specified for the new table, then all the columns in the result of the query expression are used to create same-named columns in the new table, of the corresponding data type(s). If one or more column names are specified for the new table, then the same number of columns must be present in the result of the query expression; the data types of those columns are used for the corresponding columns of the new table.

The WITH NO DATA clause specifies that the data rows which result from evaluating the query expression are not used; only the names and data types of the columns in the query result are used. The WITH NO DATA clause **must** be specified; in a future release, Derby may be modified to allow the WITH DATA clause to be provided, which would indicate that the results of the query expression should be inserted into the newly-created table. In the current release, however, only the WITH NO DATA form of the statement is accepted.

Example

```
-- create a new table using all the columns and data types
-- from an existing table:
CREATE TABLE T3 AS SELECT * FROM T1 WITH NO DATA;
-- create a new table, providing new names for the columns, but
-- using the data types from the columns of an existing table:
CREATE TABLE T3 (A,B,C,D,E) AS SELECT * FROM T1 WITH NO DATA;
-- create a new table, providing new names for the columns,
-- using the data types from the indicated columns of an existing table:
CREATE TABLE T3 (A,B,C) AS SELECT V,DP,I FROM T1 WITH NO DATA;
```

```
-- This example shows that the columns in the result of the
-- query expression may be unnamed expressions, but their data
-- types can still be used to provide the data types for the
-- corresponding named columns in the newly-created table:
CREATE TABLE T3 (X,Y) AS SELECT 2*I,2.0*F FROM T1 WITH NO DATA;
```

column-definition:

```
Simple-column-Name [ DataType ]
    [ Column-level-constraint ]*
    [ [ WITH ] DEFAULT DefaultConstantExpression
       | generated-column-spec
       | generation-clause
    ]
    [ Column-level-constraint ]*
```

The syntax of *DataType* is described in Data types. The *DataType* can be omitted only if you specify a *generation-clause*. If you omit the *DataType*, the type of the generated column is the type of the *generation-clause*. If you specify both a *DataType* and a *generation-clause*, the type of the *generation-clause* must be assignable to *DataType*.

The syntaxes of *Column-level-constraint* and *Table-level constraint* are described in CONSTRAINT clause.

Column default

For the definition of a default value, a *DefaultConstantExpression* is an expression that does not refer to any table. It can include constants, date-time special registers, current schemas, users, roles, and null:

```
DefaultConstantExpression:
          NULL
        | CURRENT { SCHEMA | SQLID }
        | USER | CURRENT_USER | SESSION_USER | CURRENT_ROLE
        | DATE
        | TIME
        | TIMESTAMP
        | CURRENT DATE | CURRENT_DATE
        | CURRENT TIME | CURRENT_TIME
        | CURRENT TIMESTAMP | CURRENT_TIMESTAMP
        | literal
```

For details about Derby *literal* values, see Data types.

The values in a *DefaultConstantExpression* must be compatible in type with the column, but a *DefaultConstantExpression* has the following additional type restrictions:

- If you specify USER, CURRENT_USER, SESSION_USER, or CURRENT_ROLE, the column must be a character column whose length is at least 8.
- If you specify CURRENT SCHEMA or CURRENT SQLID, the column must be a character column whose length is at least 128.
- If the column is an integer type, the default value must be an integer literal.
- If the column is a decimal type, the scale and precision of the default value must be within those of the column.

generated-column-spec:

```
[ GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ ( START WITH IntegerConstant
[ ,INCREMENT BY IntegerConstant] ) ]  ]  ]
```

Identity column attributes

A table can have at most one identity column.

For SMALLINT, INT, and BIGINT columns with identity attributes, Derby automatically assigns increasing integer values to the column. Identity column attributes behave like

other defaults in that when an insert statement does not specify a value for the column, Derby automatically provides the value. However, the value is not a constant; Derby automatically increments the default value at insertion time.

The IDENTITY keyword can only be specified if the data type associated with the column is one of the following exact integer types.

- SMALLINT
- INT
- BIGINT

There are two kinds of identity columns in Derby: those which are GENERATED ALWAYS and those which are GENERATED BY DEFAULT.

GENERATED ALWAYS

An identity column that is GENERATED ALWAYS will increment the default value on every insertion and will store the incremented value into the column. Unlike other defaults, you cannot insert a value directly into or update an identity column that is GENERATED ALWAYS. Instead, either specify the DEFAULT keyword when inserting into the identity column, or leave the identity column out of the insertion column list altogether. For example:

```
create table greetings
 (i int generated always as identity, ch char(50));
insert into greetings values (DEFAULT, 'hello');
insert into greetings(ch) values ('bonjour');
```

Automatically generated values in a GENERATED ALWAYS identity column are unique. Creating an identity column does not create an index on the column.

GENERATED BY DEFAULT

An identity column that is GENERATED BY DEFAULT will only increment and use the default value on insertions when no explicit value is given. Unlike GENERATED ALWAYS columns, you can specify a particular value in an insertion statement to be used instead of the generated default value.

To use the generated default, either specify the DEFAULT keyword when inserting into the identity column, or just leave the identity column out of the insertion column list. To specify a value, included it in the insertion statement. For example:

```
create table greetings
 (i int generated by default as identity, ch char(50));
-- specify value "1":
insert into greetings values (1, 'hi');
-- use generated default
insert into greetings values (DEFAULT, 'salut');
-- use generated default
insert into greetings(ch) values ('bonjour');
```

Note that unlike a GENERATED ALWAYS column, a GENERATED BY DEFAULT column does not guarantee uniqueness. Thus, in the above example, the `hi` and `salut` rows will both have an identity value of "1", because the generated column starts at "1" and the user-specified value was also "1". To prevent duplication, especially when loading or importing data, create the table using the START WITH value which corresponds to the first identity value that the system should assign. To check for this condition and disallow it, you can use a primary key or unique constraint on the GENERATED BY DEFAULT identity column.

By default, the initial value of an identity column is 1, and the amount of the increment is 1. You can specify non-default values for both the initial value and the interval amount when you define the column with the key words START WITH and INCREMENT BY. And if you specify a negative number for the increment value, Derby*decrements* the value

with each insert. If this value is positive, Derby increments the value with each insert. A value of 0 raises a statement exception.

The maximum and minimum values allowed in identity columns are determined by the data type of the column. Attempting to insert a value outside the range of values supported by the data type raises an exception.

Table 1. Maximum and Minimum Values for Columns with Generated Column Specs

Data type	Maximum Value	Minimum Value
SMALLINT	32767 (*java.lang.Short.MAX_VALUE*)	-32768 (*java.lang.Short.MIN_VALUE*)
INT	2147483647 (*java.lang.Integer.MAX_VALUE*)	-2147483648 (*java.lang.Integer.MIN_VALUE*)
BIGINT	9223372036854775807 (*java.lang.Long.MAX_VALUE*)	-9223372036854775808 (*java.lang.Long.MIN_VALUE*)

Automatically generated values in an identity column are unique. Use a primary key or unique constraint on a column to guarantee uniqueness. Creating an identity column *does not* create an index on the column.

The `IDENTITY_VAL_LOCAL` function is a non-deterministic function that returns the most recently assigned value for an identity column. See IDENTITY_VAL_LOCAL function for more information.

Note: Specify the schema, table, and column name using the same case as those names are stored in the system tables--that is, all upper case unless you used delimited identifiers when creating those database objects.

Derby keeps track of the last increment value for a column in a cache. It also stores the value of what the next increment value will be for the column on disk in the *AUTOINCREMENTVALUE* column of the *SYS.SYSCOLUMNS* system table. Rolling back a transaction does not undo this value, and thus rolled-back transactions can leave "gaps" in the values automatically inserted into an identity column. Derby behaves this way to avoid locking a row in *SYS.SYSCOLUMNS* for the duration of a transaction and keeping concurrency high.

When an insert happens within a triggered-SQL-statement, the value inserted by the triggered-SQL-statement into the identity column is available from *ConnectionInfo* only within the trigger code. The trigger code is also able to see the value inserted by the statement that caused the trigger to fire. However, the statement that caused the trigger to fire is not able to see the value inserted by the triggered-SQL-statement into the identity column. Likewise, triggers can be nested (or recursive). An SQL statement can cause trigger T1 to fire. T1 in turn executes an SQL statement that causes trigger T2 to fire. If both T1 and T2 insert rows into a table that cause Derby to insert into an identity column, trigger T1 cannot see the value caused by T2's insert, but T2 can see the value caused by T1's insert. Each nesting level can see increment values generated by itself and previous nesting levels, all the way to the top-level SQL statement that initiated the recursive triggers. You can only have 16 levels of trigger recursion.

Example

```
create table greetings
  (i int generated by default as identity (START WITH 2, INCREMENT BY 1),
  ch char(50));
-- specify value "1":
insert into greetings values (1, 'hi');
-- use generated default
```

```
insert into greetings values (DEFAULT, 'salut');
-- use generated default
insert into greetings(ch) values ('bonjour');
```

generation-clause:

```
GENERATED ALWAYS AS ( value-expression )
```

A *value-expression* is an *Expression* that resolves to a single value, with some limitations that are described here. See SQL expressions for more information about *Expressions*.

References

The *generation-clause* may reference other non-generated columns in the table, but it must not reference any generated column. The *generation-clause* must not reference a column in another table.

Functions

The *generation-clause* may invoke user-coded functions, if the functions meet the following requirements:

- The functions must not read or write SQL data.
- The functions must have been declared DETERMINISTIC.
- The functions must not invoke any of the following possibly non-deterministic system functions:
  - CURRENT_DATE
  - CURRENT_TIME
  - CURRENT_TIMESTAMP
  - CURRENT_USER
  - CURRENT_ROLE
  - CURRENT SCHEMA
  - CURRENT SQLID
  - SESSION_USER

Subqueries

The *generation-clause* must not include subqueries.

Foreign keys

If the generated column is part of a foreign key that references another table, the referential action must not specify SET NULL or SET DEFAULT, and the update rule must not specify ON UPDATE CASCADE.

Example

```
CREATE TABLE employee
(
  employeeID           int,
  name                 varchar( 50 ),
  caseInsensitiveName  GENERATED ALWAYS AS( UPPER( name ) )
);
CREATE INDEX caseInsensitiveEmployeeName ON employee( caseInsensitiveName
 );
```

CREATE TRIGGER statement

A trigger defines a set of actions that are executed when a database event occurs on a specified table. A *database event* is a delete, insert, or update operation. For example, if you define a trigger for a delete on a particular table, the trigger's action occurs whenever someone deletes a row or rows from the table.

Along with constraints, triggers can help enforce data integrity rules with actions such as cascading deletes or updates. Triggers can also perform a variety of functions such as issuing alerts, updating other tables, sending e-mail, and other useful actions.

You can define any number of triggers for a single table, including multiple triggers on the same table for the same event.

You can create a trigger in any schema where you are the schema owner. To create a trigger on a table that you do not own, you must be granted the TRIGGER privilege on that table. The database owner can also create triggers on any table in any schema.

The trigger does not need to reside in the same schema as the table on which the trigger is defined.

If a qualified trigger name is specified, the schema name cannot begin with *SYS*.

Syntax

```
CREATE TRIGGER TriggerName
{ AFTER | NO CASCADE BEFORE }
{ INSERT | DELETE | UPDATE [ OF column-Name [, column-Name]* ] }
ON table-Name
[ ReferencingClause ]
[ FOR EACH { ROW | STATEMENT } ] [ MODE DB2SQL ]
Triggered-SQL-statement
```

Before or after: when triggers fire

Triggers are defined as either *Before* or *After* triggers.

- *Before* triggers fire before the statement's changes are applied and before any constraints have been applied. Before triggers can be either row or statement triggers (see Statement versus row triggers).
- *After* triggers fire after all constraints have been satisfied and after the changes have been applied to the target table. *After* triggers can be either row or statement triggers (see Statement versus row triggers).

Insert, delete, or update: what causes the trigger to fire

A trigger is fired by one of the following database events, depending on how you define it (see Syntax above):

- INSERT
- UPDATE
- DELETE

You can define any number of triggers for a given event on a given table. For update, you can specify columns.

Referencing old and new values: the referencing clause

Many triggered-SQL-statements need to refer to data that is currently being changed by the database event that caused them to fire. The triggered-SQL-statement might need to refer to the new (post-change or "after") values.

Derby provides you with a number of ways to refer to data that is currently being changed by the database event that caused the trigger to fire. Changed data can be referred to in the triggered-SQL-statement using *transition variables* or *transition tables*. The referencing clause allows you to provide a correlation name or alias for these transition variables by specifying OLD/NEW AS *correlation-Name* .

For example, if you add the following clause to the trigger definition:

```
REFERENCING OLD AS DELETEDROW
```

you can then refer to this correlation name in the triggered-SQL-statement:

```
DELETE FROM HotelAvailability WHERE hotel_id = DELETEDROW.hotel_id
```

The OLD and NEW transition variables map to a *java.sql.ResultSet* with a single row.
Note: Only row triggers (see Statement versus row triggers) can use the transition variables. INSERT row triggers cannot reference an OLD row. DELETE row triggers cannot reference a NEW row.

For statement triggers, transition *tables* serve as a table identifier for the triggered-SQL-statement or the trigger qualification. The referencing clause allows you to provide a correlation name or alias for these transition tables by specifying OLD_TABLE/NEW_TABLE AS correlation-Name

For example:

```
REFERENCING OLD_TABLE AS DeletedHotels
```

allows you to use that new identifier (*DeletedHotels*) in the triggered-SQL-statement:

```
DELETE FROM HotelAvailability WHERE hotel_id IN
    (SELECT hotel_id FROM DeletedHotels)
```

The old and new transition tables map to a *java.sql.ResultSet* with cardinality equivalent to the number of rows affected by the triggering event.
Note: Only statement triggers (see Statement versus row triggers) can use the transition tables. INSERT statement triggers cannot reference an OLD table. DELETE statement triggers cannot reference a NEW table.

The referencing clause can designate only one new correlation or identifier and only one old correlation or identifier. Row triggers cannot designate an identifier for a transition table and statement triggers cannot designate a correlation for transition variables.

Statement versus row triggers

You have the option to specify whether a trigger is a *statement trigger* or a *row trigger*. If it is not specified in the CREATE TRIGGER statement via FOR EACH clause, then the trigger is a *statement trigger* by default.

- *statement triggers*

  A statement trigger fires once per triggering event and regardless of whether any rows are modified by the insert, update, or delete event.
- *row triggers*

  A row trigger fires once for each row affected by the triggering event. If no rows are affected, the trigger does not fire.

Note: An update that sets a column value to the value that it originally contained (for example, UPDATE T SET C = C) causes a row trigger to fire, even though the value of the column is the same as it was prior to the triggering event.

Triggered-SQL-statement

The action defined by the trigger is called the triggered-SQL-statement (in Syntax above, see the last line). It has the following limitations:

- It must not contain any dynamic parameters (?).
- It must not create, alter, or drop the table upon which the trigger is defined.
- It must not add an index to or remove an index from the table on which the trigger is defined.
- It must not add a trigger to or drop a trigger from the table upon which the trigger is defined.
- It must not commit or roll back the current transaction or change the isolation level.
- Before triggers cannot have INSERT, UPDATE or DELETE statements as their action.
- Before triggers cannot call procedures that modify SQL data as their action.
- The NEW variable of a Before trigger cannot reference a generated column.

The triggered-SQL-statement can reference database objects other than the table upon which the trigger is declared. If any of these database objects is dropped, the trigger is invalidated. If the trigger cannot be successfully recompiled upon the next execution, the invocation throws an exception and the statement that caused it to fire will be rolled back.

For more information on triggered-SQL-statements, see the *Java DB Developer's Guide*.

Order of execution

When a database event occurs that fires a trigger, Derby performs actions in this order:

- It fires *No Cascade Before* triggers.
- It performs constraint checking (primary key, unique key, foreign key, check).
- It performs the insert, update, or delete.
- It fires *After* triggers.

When multiple triggers are defined for the same database event for the same table for the same trigger time (before or after), triggers are fired in the order in which they were created.

```
-- Statements and triggers:

CREATE TRIGGER t1 NO CASCADE BEFORE UPDATE ON x
  FOR EACH ROW MODE DB2SQL
  values app.notifyEmail('Jerry', 'Table x is about to be updated');

CREATE TRIGGER FLIGHTSDELETE
  AFTER DELETE ON FLIGHTS
  REFERENCING OLD_TABLE AS DELETEDFLIGHTS
  FOR EACH STATEMENT
  DELETE FROM FLIGHTAVAILABILITY WHERE FLIGHT_ID IN
  (SELECT FLIGHT_ID FROM DELETEDFLIGHTS);

CREATE TRIGGER FLIGHTSDELETE3
  AFTER DELETE ON FLIGHTS
  REFERENCING OLD AS OLD
  FOR EACH ROW
  DELETE FROM FLIGHTAVAILABILITY WHERE FLIGHT_ID = OLD.FLIGHT_ID;
```

Note: You can find more examples in the *Java DB Developer's Guide*.

Trigger recursion

The maximum trigger recursion depth is 16.

Related information

Special system functions that return information about the current time or current user are evaluated when the trigger fires, not when it is created. Such functions include:

- CURRENT_DATE function
- CURRENT_TIME function
- CURRENT_TIMESTAMP function
- CURRENT_USER function
- SESSION_USER function
- USER function

ReferencingClause:

```
REFERENCING
{
{ OLD | NEW } [ ROW ] [ AS ] correlation-Name [ { OLD | NEW } [ ROW ] [
 AS ] correlation-Name ] |
{ OLD TABLE | NEW TABLE } [ AS ] Identifier [ { OLD TABLE | NEW TABLE }
 [AS] Identifier ]  |
{ OLD_TABLE | NEW_TABLE } [ AS ] Identifier [ { OLD_TABLE | NEW_TABLE }
 [AS] Identifier ]
}
```

Note: The **OLD_TABLE | NEW_TABLE** syntax is deprecated since it is not SQL compliant and is intended for backward compatibility and DB2 compatibility.

CREATE VIEW statement

Views are virtual tables formed by a query. A view is a dictionary object that you can use until you drop it. Views are not updatable.

If a qualified view name is specified, the schema name cannot begin with *SYS*.

The view owner automatically gains the SELECT privilege on the view. The SELECT privilege cannot be revoked from the view owner. The database owner automatically gains the SELECT privilege on the view and is able to grant this privilege to other users. The SELECT privilege cannot be revoked from the database owner.

The view owner can only grant the SELECT privilege to other users if the view owner also owns the underlying objects.

If the underlying objects that the view references are not owned by the view owner, the view owner must be granted the appropriate privileges. For example, if the authorization ID `user2` attempts to create a view called `user2.v2` that references table `user1.t1` and function `user1.f_abs()`, then `user2` must have the SELECT privilege on table `user1.t1` and the EXECUTE privilege on function `user1.f_abs()`.

The privilege to grant the SELECT privilege cannot be revoked. If a required privilege on one of the underlying objects that the view references is revoked, then the view is dropped.

Syntax

```
CREATE VIEW view-Name
    [ ( Simple-column-Name [, Simple-column-Name] * ) ]
AS Query
```

A view definition can contain an optional view column list to explicitly name the columns in the view. If there is no column list, the view inherits the column names from the underlying query. All columns in a view must be uniquely named.

```
CREATE VIEW SAMP.V1 (COL_SUM, COL_DIFF)
 AS SELECT COMM + BONUS, COMM - BONUS
 FROM SAMP.EMPLOYEE;

CREATE VIEW SAMP.VEMP_RES (RESUME)
 AS VALUES 'Delores M. Quintana', 'Heather A. Nicholls', 'Bruce Adamson';

CREATE VIEW SAMP.PROJ_COMBO
 (PROJNO, PRENDATE, PRSTAFF, MAJPROJ)
 AS SELECT PROJNO, PRENDATE, PRSTAFF, MAJPROJ
 FROM SAMP.PROJECT UNION ALL
SELECT PROJNO, EMSTDATE, EMPTIME, EMPNO
 FROM SAMP.EMP_ACT
 WHERE EMPNO IS NOT NULL;
```

Statement dependency system

View definitions are dependent on the tables and views referenced within the view definition. DML (data manipulation language) statements that contain view references depend on those views, as well as the objects in the view definitions that the views are dependent on. Statements that reference the view depend on indexes the view uses; which index a view uses can change from statement to statement based on how the query is optimized. For example, given:

```
CREATE TABLE T1 (C1 DOUBLE PRECISION);

CREATE FUNCTION SIN (DATA DOUBLE)
 RETURNS DOUBLE EXTERNAL NAME 'java.lang.Math.sin'
 LANGUAGE JAVA PARAMETER STYLE JAVA;
```

```
CREATE VIEW V1 (C1) AS SELECT SIN(C1) FROM T1;
```

the following SELECT:

```
SELECT * FROM V1
```

is dependent on view *V1*, table *T1,* and external scalar function *SIN.*

DECLARE GLOBAL TEMPORARY TABLE statement

The DECLARE GLOBAL TEMPORARY TABLE statement defines a temporary table for the current connection.

These tables do not reside in the system catalogs and are not persistent. Temporary tables exist only during the connection that declared them and cannot be referenced outside of that connection. When the connection closes, the rows of the table are deleted, and the in-memory description of the temporary table is dropped.

Temporary tables are useful when:

- The table structure is not known before using an application.
- Other users do not need the same table structure.
- Data in the temporary table is needed while using the application.
- The table can be declared and dropped without holding the locks on the system catalog.

Syntax

```
DECLARE GLOBAL TEMPORARY TABLE table-Name
    { column-definition [ , column-definition ] * }
[ ON COMMIT {DELETE | PRESERVE} ROWS ]
NOT LOGGED [ON ROLLBACK DELETE ROWS]
```

table-Name

Names the temporary table. If a schema-Name other than SESSION is specified, an error will occur (SQLSTATE 428EK). If the schema-Name is not specified, SESSION is assigned. Multiple connections can define declared global temporary tables with the same name because each connection has its own unique table descriptor for it.

Using SESSION as the schema name of a physical table will not cause an error, but is discouraged. The SESSION schema name should be reserved for the temporary table schema.

column-definition

See column-definition for `CREATE TABLE` for more information on `column-definition`. `DECLARE GLOBAL TEMPORARY TABLE` does not allow `generated-column-spec` in the `column-definition`.

Data type

Supported data types are:

- BIGINT
- CHAR
- DATE
- DECIMAL
- DOUBLE
- DOUBLE PRECISION
- FLOAT
- INTEGER
- NUMERIC

- REAL
- SMALLINT
- TIME
- TIMESTAMP
- VARCHAR

ON COMMIT

Specifies the action taken on the global temporary table when a COMMIT operation is performed.

DELETE ROWS

All rows of the table will be deleted if no hold-able cursor is open on the table. This is the default value for ON COMMIT. If you specify ON ROLLBACK DELETE ROWS, this will delete all the rows in the table only if the temporary table was used. ON COMMIT DELETE ROWS will delete the rows in the table even if the table was not used (if the table does not have hold-able cursors open on it).

PRESERVE ROWS

The rows of the table will be preserved.

NOT LOGGED

Specifies the action taken on the global temporary table when a rollback operation is performed. When a ROLLBACK (or ROLLBACK TO SAVEPOINT) operation is performed, if the table was created in the unit of work (or savepoint), the table will be dropped. If the table was dropped in the unit of work (or savepoint), the table will be restored with no rows.

ON ROLLBACK DELETE ROWS

This is the default value for NOT LOGGED. NOT LOGGED [ON ROLLBACK DELETE ROWS]] specifies the action that is to be taken on the global temporary table when a ROLLBACK or (ROLLBACK TO SAVEPOINT) operation is performed. If the table data has been changed, all the rows will be deleted.

Examples

```
set schema myapp;

create table t1(c11 int, c12 date);

declare global temporary table SESSION.t1(c11 int) not logged;
-- The SESSION qualification is redundant here because temporary
-- tables can only exist in the SESSION schema.

declare global temporary table t2(c21 int) not logged;
-- The temporary table is not qualified here with SESSION because
 temporary
-- tables can only exist in the SESSION schema.

insert into SESSION.t1 values (1);
-- SESSION qualification is mandatory here if you want to use
-- the temporary table, because the current schema is "myapp."

select * from t1;
-- This select statement is referencing the "myapp.t1" physical
-- table since the table was not qualified by SESSION.
```

Note: Temporary tables can be declared only in the SESSION schema. You should never declare a physical schema with the SESSION name.
The following is a list of DB2 UDB DECLARE GLOBAL TEMPORARY TABLE functions that are not supported by Derby:

- IDENTITY column-options
- IDENTITY attribute in copy-options
- AS (fullselect) DEFINITION ONLY
- NOT LOGGED ON ROLLBACK PRESERVE ROWS
- IN tablespace-name
- PARTITIONING KEY
- WITH REPLACE

Restrictions on Declared Global Temporary Tables

Derby does not support the following features on temporary tables. Some of these features are specific to temporary tables and some are specific to Derby.

Temporary tables cannot be specified in the following statements:

- ALTER TABLE
- CREATE INDEX
- CREATE SYNONYM
- CREATE TRIGGER
- CREATE VIEW
- GRANT
- LOCK TABLE
- RENAME
- REVOKE

You cannot use the following features with temporary tables:

- Synonyms, triggers and views on SESSION schema tables (including physical tables and temporary tables)
- Caching statements that reference SESSION schema tables and views
- Temporary tables cannot be specified in referential constraints and primary keys
- Temporary tables cannot be referenced in a triggered-SQL-statement
- Check constraints on columns
- Generated-column-spec
- Importing into temporary tables

If a statement that performs an insert, update, or delete to the temporary table encounters an error, all the rows of the temporary table are deleted.

The following data types cannot be used with Declared Global Temporary Tables:

- BLOB
- CHAR FOR BIT DATA
- CLOB
- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA
- VARCHAR FOR BIT DATA
- XML

DELETE statement

Syntax

```
{
    DELETE FROM table-Name [[AS] correlation-Name]
        [WHERE clause] |
    DELETE FROM table-Name WHERE CURRENT OF
}
```

The first syntactical form, called a searched delete, removes all rows identified by the table name and WHERE clause.

The second syntactical form, called a positioned delete, deletes the current row of an open, updatable cursor. For more information about updatable cursors, see SELECT statement.

Examples

```
DELETE FROM SAMP.IN_TRAY

stmt.executeUpdate("DELETE FROM SAMP.IN_TRAY WHERE CURRENT OF " +
 resultSet.getCursorName());
```

Statement dependency system

A searched delete statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), and any other table named in the WHERE clause. A CREATE or DROP INDEX statement for the target table of a prepared searched delete statement invalidates the prepared searched delete statement.

The positioned delete statement depends on the cursor and any tables the cursor references. You can compile a positioned delete even if the cursor has not been opened yet. However, removing the open cursor with the JDBC *close* method invalidates the positioned delete.

A CREATE or DROP INDEX statement for the target table of a prepared positioned delete invalidates the prepared positioned delete statement.

DROP statements

Use DROP statements to remove functions, indexes, procedures, roles, schemas, synonyms, tables, triggers, and views.

DROP FUNCTION statement

Syntax

```
DROP FUNCTION function-name
```

Identifies the particular function to be dropped, and is valid only if there is exactly one function instance with the *function-name* in the schema. The identified function can have any number of parameters defined for it.

An error will occur in any of the following circumstances:

- If no function with the indicated name exists in the named or implied schema (the error is SQLSTATE 42704)
- If there is more than one specific instance of the function in the named or implied schema
- If you try to drop a user-defined function that is invoked in the *generation-clause* of a generated column
- If you try to drop a user-defined function that is invoked in a view

DROP INDEX statement

DROP INDEX removes the specified index.

Syntax

```
DROP INDEX index-Name
```

```
DROP INDEX OrigIndex

DROP INDEX DestIndex
```

Statement dependency system

If there is an open cursor on the table from which the index is dropped, the DROP INDEX statement generates an error and does not drop the index. Otherwise, statements that depend on the index's table are invalidated.

DROP PROCEDURE statement

Syntax

```
DROP PROCEDURE procedure-Name
```

Identifies the particular procedure to be dropped, and is valid only if there is exactly one procedure instance with the *procedure-name* in the schema. The identified procedure can have any number of parameters defined for it. If no procedure with the indicated name in the named or implied schema, an error (SQLSTATE 42704) will occur. An error will also occur if there is more than one specific instance of the procedure in the named or implied schema.

DROP ROLE statement

The DROP ROLE statement allows you to drop an SQL role.

Only the database owner can drop a role.

For more information on roles, see "Using SQL roles" in the *Java DB Developer's Guide*.

Syntax

```
DROP ROLE roleName
```

Dropping a role has the effect of removing the role from the database dictionary. This means that no session user can henceforth set that role (see SET ROLE statement), and any existing sessions that have that role as the current role (see CURRENT_ROLE function) will now have a NULL CURRENT_ROLE. Dropping a role also has the effect of revoking that role from any user and role it has been granted to. See REVOKE statement for information on how revoking a role may impact any dependent objects.

Example

```
DROP ROLE reader;
```

DROP SCHEMA statement

The DROP SCHEMA statement drops a schema. The target schema must be empty for the drop to succeed.

Neither the *APP* schema (the default user schema) nor the *SYS* schema can be dropped.

Syntax

```
DROP SCHEMA schemaName RESTRICT
```

The RESTRICT keyword enforces the rule that no objects can be defined in the specified schema for the schema to be deleted from the database. The RESTRICT keyword is required

```
-- Drop the SAMP schema
-- The SAMP schema may only be deleted from the database
-- if no objects are defined in the SAMP schema.

DROP SCHEMA SAMP RESTRICT
```

DROP SYNONYM statement

Drops the specified synonym from a table or view.

Syntax

```
DROP SYNONYM synonym-Name
```

DROP TABLE statement

DROP TABLE removes the specified table.

Syntax

```
DROP TABLE table-Name
```

Statement dependency system

Triggers, constraints (primary, unique, check and references from the table being dropped) and indexes on the table are silently dropped. The existence of an open cursor that references table being dropped cause the DROP TABLE statement to generate an error, and the table is not dropped.

Dropping a table invalidates statements that depend on the table. (Invalidating a statement causes it to be recompiled upon the next execution. See Interaction with the dependency system.)

DROP TRIGGER statement

DROP TRIGGER removes the specified trigger.

Syntax

```
DROP TRIGGER TriggerName
```

```
DROP TRIGGER TRIG1
```

Statement dependency system

When a table is dropped, all triggers on that table are automatically dropped. (You don't have to drop a table's triggers before dropping the table.)

DROP VIEW statement

Drops the specified view.

Syntax

```
DROP VIEW view-Name
```

```
DROP VIEW AnIdentifier
```

Statement dependency system

Any statements referencing the view are invalidated on a DROP VIEW statement. DROP VIEW is disallowed if there are any views or open cursors dependent on the view. The view must be dropped before any objects that it is dependent on can be dropped.

GRANT statement

Use the GRANT statement to give permissions to a specific user or role, or to all users, to perform actions on database objects. You can also use the GRANT statement to grant a role to a user, to PUBLIC, or to another role.

The following types of permissions can be granted:
- Delete data from a specific table.
- Insert data into a specific table.
- Create a foreign key reference to the named table or to a subset of columns from a table.
- Select data from a table, view, or a subset of columns in a table.
- Create a trigger on a table.

- Update data in a table or in a subset of columns in a table.
- Run a specified function or procedure.

Before you issue a GRANT statement, check that the `derby.database.sqlAuthorization` property is set to `true`. The `derby.database.sqlAuthorization` property enables the SQL Authorization mode.

You can grant privileges to database objects that you are authorized to grant. See the CREATE statement for the database object that you want to grant privileges on for more information.

The syntax that you use for the GRANT statement depends on whether you are granting privileges to a table or to a routine, or granting a role.

For more information on using the GRANT statement, see "Using SQL standard authorization" in the *Java DB Developer's Guide*.

Syntax for tables

```
GRANT privilege-type ON [TABLE] { table-Name | view-Name } TO grantees
```

Syntax for routines

```
GRANT EXECUTE ON { FUNCTION | PROCEDURE } routine-designator TO grantees
```

Syntax for roles

```
GRANT roleName [ {, roleName }* ] TO grantees
```

Before you can grant a role to a user or to another role, you must create the role using the CREATE ROLE statement. Only the database owner can grant a role.

A role A *contains* another role B if role B is granted to role A, or is contained in a role C granted to role A. Privileges granted to a contained role are inherited by the containing roles. So the set of privileges identified by role A is the union of the privileges granted to role A and the privileges granted to any contained roles of role A.

privilege-types

```
  ALL PRIVILEGES |
  privilege-list
```

privilege-list

```
 table-privilege {, table-privilege }*
```

table-privilege

```
  DELETE |
  INSERT |
  REFERENCES [column list] |
  SELECT [column list] |
  TRIGGER |
  UPDATE [column list]
```

column list

```
  ( column-identifier {, column-identifier}* )
```

Use the ALL PRIVILEGES privilege type to grant all of the permissions to the user or role for the specified table. You can also grant one or more table privileges by specifying a privilege-list.

Use the DELETE privilege type to grant permission to delete rows from the specified table.

Use the INSERT privilege type to grant permission to insert rows into the specified table.

Use the REFERENCES privilege type to grant permission to create a foreign key reference to the specified table. If a column list is specified with the REFERENCES privilege, the permission is valid on only the foreign key reference to the specified columns.

Use the SELECT privilege type to grant permission to perform SELECT statements on a table or view. If a column list is specified with the SELECT privilege, the permission is valid on only those columns. If no column list is specified, then the privilege is valid on all of the columns in the table.

Use the TRIGGER privilege type to grant permission to create a trigger on the specified table.

Use the UPDATE privilege type to grant permission to use the UPDATE statement on the specified table. If a column list is specified, the permission applies only to the specified columns. To update a row using a statement that includes a WHERE clause, you must have SELECT permission on the columns in the row that you want to update.

grantees

```
{ AuthorizationIdentifier | roleName | PUBLIC }
[, { AuthorizationIdentifier | roleName | PUBLIC } ] *
```

You can grant privileges or roles to specific users or roles or to all users. Use the keyword PUBLIC to specify all users. When PUBLIC is specified, the privileges or roles affect all current and future users. The privileges granted to PUBLIC and to individual users or roles are independent privileges. For example, a SELECT privilege on table `t` is granted to both PUBLIC and to the authorization ID `harry`. The SELECT privilege is later revoked from the authorization ID `harry`, but Harry can access the table `t` through the PUBLIC privilege.

Either the object owner or the database owner can grant privileges to a user or to a role. Only the database owner can grant a role to a user or to another role.

routine-designator

```
{
 function-name | procedure-name
}
```

Examples

To grant the SELECT privilege on table `t` to the authorization IDs `maria` and `harry`, use the following syntax:

```
GRANT SELECT ON TABLE t TO maria,harry
```

To grant the UPDATE and TRIGGER privileges on table `t` to the authorization IDs `anita` and `zhi`, use the following syntax:

```
GRANT UPDATE, TRIGGER ON TABLE t TO anita,zhi
```

To grant the SELECT privilege on table `s.v` to all users, use the following syntax:

```
GRANT SELECT ON TABLE s.v to PUBLIC
```

To grant the EXECUTE privilege on procedure p to the authorization ID `george`, use the following syntax:

```
GRANT EXECUTE ON PROCEDURE p TO george
```

To grant the role `purchases_reader_role` to the authorization IDs `george` and `maria`, use the following syntax:

```
GRANT purchases_reader_role TO george,maria
```

To grant the SELECT privilege on table `t` to the role `purchases_reader_role`, use the following syntax:

```
GRANT SELECT ON TABLE t TO purchases_reader_role
```

INSERT statement

An INSERT statement creates a row or rows and stores them in the named table. The number of values assigned in an INSERT statement must be the same as the number of specified or implied columns.

Whenever you insert into a table which has generated columns, Derby calculates the values of those columns.

Syntax

```
INSERT INTO table-Name
    [ (Simple-column-Name [ , Simple-column-Name]* ) ]
   Query
```

Query can be:
- a *SelectExpression*
- a VALUES list
- a multiple-row VALUES expression

  Single-row and multiple-row lists can include the keyword DEFAULT. Specifying DEFAULT for a column inserts the column's default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table. For more information see VALUES Expression.

  The DEFAULT literal is the only value which you can directly insert into a generated column.
- UNION expressions

For more information about Query, see Query.

```
INSERT INTO COUNTRIES
      VALUES ('Taiwan', 'TW', 'Asia')

-- Insert a new department into the DEPARTMENT table,
-- but do not assign a manager to the new department
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
 VALUES ('E31', 'ARCHITECTURE', 'E01')
-- Insert two new departments using one statement
-- into the DEPARTMENT table as in the previous example,
-- but do not assign a manager to the new department.
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
 VALUES ('B11', 'PURCHASING', 'B01'),
 ('E41', 'DATABASE ADMINISTRATION', 'E01')
--  Create a temporary table MA_EMP_ACT with the
-- same columns as the EMP_ACT table.
-- Load MA_EMP_ACT with the rows from the EMP_ACT
-- table with a project number (PROJNO)
-- starting with the letters 'MA'.
CREATE TABLE MA_EMP_ACT
    (
 EMPNO CHAR(6)  NOT NULL,
 PROJNO CHAR(6)  NOT NULL,
 ACTNO SMALLINT  NOT NULL,
 EMPTIME DEC(5,2),
 EMSTDATE DATE,
 EMENDATE  DATE
```

```
    );

INSERT INTO MA_EMP_ACT
 SELECT * FROM EMP_ACT
 WHERE SUBSTR(PROJNO, 1, 2) = 'MA';
-- Insert the DEFAULT value for the LOCATION column
INSERT INTO DEPARTMENT
      VALUES ('E31', 'ARCHITECTURE', '00390', 'E01', DEFAULT)
```

Statement dependency system

The INSERT statement depends on the table being inserted into, all of the conglomerates (units of storage such as heaps or indexes) for that table, and any other table named in the statement. Any statement that creates or drops an index or a constraint for the target table of a prepared INSERT statement invalidates the prepared INSERT statement.

LOCK TABLE statement

The LOCK TABLE statement allows you to explicitly acquire a shared or exclusive table lock on the specified table. The table lock lasts until the end of the current transaction.

To lock a table, you must either be the database owner or the table owner.

Explicitly locking a table is useful to:

- Avoid the overhead of multiple row locks on a table (in other words, user-initiated lock escalation)
- Avoid deadlocks

You cannot lock system tables with this statement.

Syntax

```
LOCK TABLE table-Name IN { SHARE | EXCLUSIVE } MODE
```

After a table is locked in either mode, a transaction does not acquire any subsequent row-level locks on a table. For example, if a transaction locks the entire `Flights` table in share mode in order to read data, a particular statement might need to lock a particular row in exclusive mode in order to update the row. However, the previous table-level lock on the `Flights` table forces the exclusive lock to be table-level as well.

If the specified lock cannot be acquired because another connection already holds a lock on the table, a statement-level exception is raised (*SQLState* X0X02) after the deadlock timeout period.

Examples

To lock the entire `Flights` table in share mode to avoid a large number of row locks, use the following statement:

```
LOCK TABLE Flights IN SHARE MODE;
SELECT *
FROM Flights
WHERE orig_airport > 'OOO';
```

You have a transaction with multiple UPDATE statements. Since each of the individual statements acquires only a few row-level locks, the transaction will not automatically upgrade the locks to a table-level lock. However, collectively the UPDATE statements acquire and release a large number of locks, which might result in deadlocks. For this type of transaction, you can acquire an exclusive table-level lock at the beginning of the transaction. For example:

```
LOCK TABLE FlightAvailability IN EXCLUSIVE MODE;
UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-03-31');
```

```
UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-11');

UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-12');

UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-15');
```

If a transaction needs to look at a table before updating the table, acquire an exclusive lock before selecting to avoid deadlocks. For example:

```
LOCK TABLE Maps IN EXCLUSIVE MODE;
SELECT MAX(map_id) + 1 FROM Maps;
-- INSERT INTO Maps . . .
```

RENAME statements

Use the Rename statements with indexes, columns, and tables.

RENAME COLUMN statement

Use the RENAME COLUMN statement to rename a column in a table.

The RENAME COLUMN statement allows you to rename an existing column in an existing table in any schema (except the schema *SYS*).

To rename a column, you must either be the database owner or the table owner.

Other types of table alterations are possible; see ALTER TABLE statement for more information.

Syntax

```
RENAME COLUMN table-Name.simple-Column-Name TO simple-Column-Name
```

Examples

To rename the *manager* column in table *employee* to *supervisor*, use the following syntax:

```
RENAME COLUMN EMPLOYEE.MANAGER TO SUPERVISOR
```

You can combine ALTER TABLE and RENAME COLUMN to modify a column's data type. To change column *c1* of table *t* to the new data type *NEWTYPE*:

```
    ALTER TABLE t ADD COLUMN c1_newtype NEWTYPE
    UPDATE t SET c1_newtype = c1
    ALTER TABLE t DROP COLUMN c1
    RENAME COLUMN t.c1_newtype TO c1
```

Usage notes

Restriction: If a view, trigger, check constraint, foreign key constraint, or *generation-clause* of a generated column references the column, an attempt to rename it will generate an error.

Restriction: The RENAME COLUMN statement is not allowed if there are any open cursors that reference the column that is being altered.

Note: If there is an index defined on the column, the column can still be renamed; the index is automatically updated to refer to the column by its new name

RENAME INDEX statement

This statement allows you to rename an index in the current schema. Users cannot rename indexes in the *SYS* schema.

Syntax

```
RENAME INDEX index-Name TO new-index-Name
```

```
RENAME INDEX DESTINDEX TO ARRIVALINDEX
```

Statement dependency system

RENAME INDEX is not allowed if there are any open cursors that reference the index being renamed.

RENAME TABLE statement

RENAME TABLE allows you to rename an existing table in any schema (except the schema *SYS*).

To rename a table, you must either be the database owner or the table owner.

Syntax

```
RENAME TABLE table-Name TO new-Table-Name
```

If there is a view or foreign key that references the table, attempts to rename it will generate an error. In addition, if there are any check constraints or triggers on the table, attempts to rename it will also generate an error.

```
RENAME TABLE SAMP.EMP_ACT TO EMPLOYEE_ACT
```

Also see ALTER TABLE statement for more information.

Statement dependency system

If there is an index defined on the table, the table can be renamed.

The RENAME TABLE statement is not allowed if there are any open cursors that reference the table that is being altered.

REVOKE statement

Use the REVOKE statement to remove permissions from a specific user or role, or from all users, to perform actions on database objects. You can also use the REVOKE statement to revoke a role from a user, from PUBLIC, or from another role.

The following types of permissions can be revoked:
- Delete data from a specific table.
- Insert data into a specific table.
- Create a foreign key reference to the named table or to a subset of columns from a table.
- Select data from a table, view, or a subset of columns in a table.
- Create a trigger on a table.
- Update data in a table or in a subset of columns in a table.
- Run a specified routine (function or procedure).

The `derby.database.sqlAuthorization` property must be set to `true` before you can use the GRANT statement or the REVOKE statement. The `derby.database.sqlAuthorization` property enables SQL Authorization mode.

You can revoke privileges for an object if you are the owner of the object or the database owner.

The syntax that you use for the REVOKE statement depends on whether you are revoking privileges to a table or to a routine, or whether you are revoking a role.

For more information on using the REVOKE statement, see "Using SQL standard authorization" in the *Java DB Developer's Guide*.

Syntax for tables

```
REVOKE privilege-type ON [ TABLE ] { table-Name | view-Name } FROM
 grantees
```

Revoking a privilege without specifying a column list revokes the privilege for all of the columns in the table.

Syntax for routines

```
REVOKE EXECUTE ON { FUNCTION | PROCEDURE } routine-designator FROM
 grantees RESTRICT
```

You must use the RESTRICT clause on REVOKE statements for routines. The RESTRICT clause specifies that the EXECUTE privilege cannot be revoked if the specified routine is used in a view, trigger, or constraint, and the privilege is being revoked from the owner of the view, trigger, or constraint.

Syntax for roles

```
REVOKE roleName [ {, roleName }* ] FROM grantees
```

Only the database owner can revoke a role.

privilege-types

```
  ALL PRIVILEGES |
  privilege-list
```

privilege-list

```
  table-privilege {, table-privilege }*
```

table-privilege

```
  DELETE |
  INSERT |
  REFERENCES [column list] |
  SELECT [column list] |
  TRIGGER |
  UPDATE [column list]
```

column list

```
  ( column-identifier {, column-identifier}* )
```

Use the ALL PRIVILEGES privilege type to revoke all of the permissions from the user or role for the specified table. You can also revoke one or more table privileges by specifying a privilege-list.

Use the DELETE privilege type to revoke permission to delete rows from the specified table.

Use the INSERT privilege type to revoke permission to insert rows into the specified table.

Use the REFERENCES privilege type to revoke permission to create a foreign key reference to the specified table. If a column list is specified with the REFERENCES privilege, the permission is revoked on only the foreign key reference to the specified columns.

Use the SELECT privilege type to revoke permission to perform SELECT statements on a table or view. If a column list is specified with the SELECT privilege, the permission is revoked on only those columns. If no column list is specified, then the privilege is valid on all of the columns in the table.

Use the TRIGGER privilege type to revoke permission to create a trigger on the specified table.

Use the UPDATE privilege type to revoke permission to use the UPDATE statement on the specified table. If a column list is specified, the permission is revoked only on the specified columns.

grantees

```
{ AuthorizationIdentifier | roleName | PUBLIC }
[,{ AuthorizationIdentifier | roleName | PUBLIC } ] *
```

You can revoke the privileges from specific users or roles or from all users. Use the keyword PUBLIC to specify all users. The privileges revoked from PUBLIC and from individual users or roles are independent privileges. For example, a SELECT privilege on table `t` is granted to both PUBLIC and to the authorization ID `harry`. The SELECT privilege is later revoked from the authorization ID `harry`, but the authorization ID `harry` can access the table `t` through the PUBLIC privilege.

You can revoke a role from a role, from a user, or from PUBLIC.

Restriction: You cannot revoke the privileges of the owner of an object.

routine-designator

```
 {
  qualified-name [ signature ]
 }
```

Prepared statements and open result sets/cursors

Checking for privileges happens at statement execution time, so prepared statements are still usable after a revoke action. If sufficient privileges are still available for the session, prepared statements will be executed, and for queries, a result set will be returned.

Once a result set has been returned to the application (by executing a prepared statement or by direct execution), it will remain accessible even if privileges or roles are revoked in a way that would cause another execution of the same statement to fail.

Cascading object dependencies

For views, triggers, and constraints, if the privilege on which the object depends on is revoked, the object is automatically dropped. Derby does not try to determine if you have other privileges that can replace the privileges that are being revoked. For more information, see "SQL standard authorization" in the *Java DB Developer's Guide*.

Limitations

The following limitations apply to the REVOKE statement:

Table-level privileges

All of the table-level privilege types for a specified grantee and table ID are stored in one row in the SYSTABLEPERMS system table. For example, when `user2` is granted the SELECT and DELETE privileges on table `user1.t1`, a row is added to the SYSTABLEPERMS table. The GRANTEE field contains `user2` and the TABLEID contains `user1.t1`. The SELECTPRIV and DELETEPRIV fields are set to Y. The remaining privilege type fields are set to N.

When a grantee creates an object that relies on one of the privilege types, the Derby engine tracks the dependency of the object on the specific row in the

SYSTABLEPERMS table. For example, user2 creates the view v1 by using the statement SELECT * FROM user1.t1, the dependency manager tracks the dependency of view v1 on the row in SYSTABLEPERMS for GRANTEE(user2), TABLEID(user1.t1). The dependency manager knows only that the view is dependent on a privilege type in that specific row, but does not track exactly which privilege type the view is dependent on.

When a REVOKE statement for a table-level privilege is issued for a grantee and table ID, all of the objects that are dependent on the grantee and table ID are dropped. For example, if user1 revokes the DELETE privilege on table t1 from user2, the row in SYSTABLEPERMS for GRANTEE(user2), TABLEID(user1.t1) is modified by the REVOKE statement. The dependency manager sends a revoke invalidation message to the view user2.v1 and the view is dropped even though the view is not dependent on the DELETE privilege for GRANTEE(user2), TABLEID(user1.t1).

Column-level privileges

Only one type of privilege for a specified grantee and table ID are stored in one row in the SYSCOLPERMS system table. For example, when user2 is granted the SELECT privilege on table user1.t1 for columns c12 and c13, a row is added to the SYSCOLPERMS. The GRANTEE field contains user2, the TABLEID contains user1.t1, the TYPE field contains S, and the COLUMNS field contains c12, c13.

When a grantee creates an object that relies on the privilege type and the subset of columns in a table ID, the Derby engine tracks the dependency of the object on the specific row in the SYSCOLPERMS table. For example, user2 creates the view v1 by using the statement SELECT c11 FROM user1.t1, the dependency manager tracks the dependency of view v1 on the row in SYSCOLPERMS for GRANTEE(user2), TABLEID(user1.t1), TYPE(S). The dependency manager knows that the view is dependent on the SELECT privilege type, but does not track exactly which columns the view is dependent on.

When a REVOKE statement for a column-level privilege is issued for a grantee, table ID, and type, all of the objects that are dependent on the grantee, table ID, and type are dropped. For example, if user1 revokes the SELECT privilege on column c12 on table user1.t1 from user2, the row in SYSCOLPERMS for GRANTEE(user2), TABLEID(user1.t1), TYPE(S) is modified by the REVOKE statement. The dependency manager sends a revoke invalidation message to the view user2.v1 and the view is dropped even though the view is not dependent on the column c12 for GRANTEE(user2), TABLEID(user1.t1), TYPE(S).

Roles

Derby tracks any dependencies on the definer's current role for views, constraints, and triggers. If privileges were obtainable only via the current role when the object in question was defined, that object depends on the current role. The object will be dropped if the role is revoked from the defining user or from PUBLIC, as the case may be. Also, if a contained role of the current role in such cases is revoked, dependent objects will be dropped. Note that dropping may be too pessimistic. This is because Derby does not currently make an attempt to recheck if the necessary privileges are still available in such cases.

Revoke examples

To revoke the SELECT privilege on table t from the authorization IDs maria and harry, use the following syntax:

```
REVOKE SELECT ON TABLE t FROM maria,harry
```

To revoke the UPDATE and TRIGGER privileges on table t from the authorization IDs anita and zhi, use the following syntax:

```
REVOKE UPDATE, TRIGGER ON TABLE t FROM anita,zhi
```

To revoke the SELECT privilege on table `s.v` from all users, use the following syntax:

```
REVOKE SELECT ON TABLE s.v FROM PUBLIC
```

To revoke the UPDATE privilege on columns `c1` and `c2` of table `s.v` from all users, use the following syntax:

```
REVOKE UPDATE (c1,c2) ON TABLE s.v FROM PUBLIC
```

To revoke the EXECUTE privilege on procedure p from the authorization ID `george`, use the following syntax:

```
REVOKE EXECUTE ON PROCEDURE p FROM george RESTRICT
```

To revoke the role `purchases_reader_role` from the authorization IDs `george` and `maria`, use the following syntax:

```
REVOKE purchases_reader_role FROM george,maria
```

To revoke the SELECT privilege on table `t` from the role `purchases_reader_role`, use the following syntax:

```
REVOKE SELECT ON TABLE t FROM purchases_reader_role
```

SET statements

Use the SET statements to set the current role, schema, or isolation level.

SET ISOLATION statement

The SET ISOLATION statement allows a user to change the isolation level for the user's connection. Valid levels are SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED.

Issuing this statement always commits the current transaction. The JDBC *java.sql.Connection.setTransactionIsolation* method behaves almost identically to this command, with one exception: if you are using the embedded driver, and if the call to *java.sql.Connection.setTransactionIsolation* does not actually change the isolation level (that is, if it sets the isolation level to its current value), the current transaction is not committed.

For information about isolation levels, see "Locking, concurrency, and isolation" in the *Java DB Developer's Guide*. For information about the JDBC *java.sql.Connection.setTransactionIsolation* method, see java.sql.Connection.setTransactionIsolation method.

Syntax

```
SET [ CURRENT ] ISOLATION [ = ]
{
UR | DIRTY READ | READ UNCOMMITTED
CS | READ COMMITTED | CURSOR STABILITY
RS |
RR | REPEATABLE READ | SERIALIZABLE
RESET
}
```

```
set isolation serializable;
```

SET ROLE statement

The SET ROLE statement allows you to set the current role for the current SQL context of a session.

You can set a role only if the current user has been granted the role, or if the role has been granted to PUBLIC.

For more information on roles, see "Using SQL roles" in the *Java DB Developer's Guide*.

Syntax

```
SET ROLE { roleName | 'string-constant' | ? | NONE }
```

If you specify a *roleName* of NONE, the effect is to unset the current role.

If you specify the role as a string constant or as a dynamic parameter specification (?), any leading and trailing blanks are trimmed from the string before attempting to use the remaining (sub)string as a *roleName*. The dynamic parameter specification can be used in prepared statements, so the SET ROLE statement can be prepared once and then executed with different role values. You cannot specify NONE as a dynamic parameter.

Setting a role identifies a set of privileges that is a union of the following:

- The privileges granted to that role
- The union of privileges of roles contained in that role (for a definition of role containment, see "Syntax for roles" in GRANT statement)

In a session, the *current privileges* define what the session is allowed to access. The *current privileges* are the union of the following:

- The privileges granted to the current user
- The privileges granted to PUBLIC
- The privileges identified by the current role, if set

The SET ROLE statement is not transactional; a rollback does not undo the effect of setting a role. If a transaction is in progress, an attempt to set a role results in an error.

Examples

```
SET ROLE reader;
```

```
  // These examples show the use of SET ROLE in JDBC statements.
  // The case normal form is visible in the SYS.SYSROLES system table.
  stmt.execute("SET ROLE admin");      -- case normal form: ADMIN
  stmt.execute("SET ROLE \"admin\"");  -- case normal form: admin
  stmt.execute("SET ROLE none");       -- special case

  PreparedStatement ps = conn.prepareStatement("SET ROLE ?");
  ps.setString(1, "  admin ");  -- on execute: case normal form: ADMIN
  ps.setString(1, "\"admin\""); -- on execute: case normal form: admin
  ps.setString(1, "none");      -- on execute: syntax error
  ps.setString(1, "\"none\"");  -- on execute: case normal form: none
```

SET SCHEMA statement

The SET SCHEMA statement sets the default schema for a connection's session to the designated schema. The default schema is used as the target schema for all statements issued from the connection that do not explicitly specify a schema name.

The target schema must exist for the SET SCHEMA statement to succeed. If the schema doesn't exist an error is returned. See CREATE SCHEMA statement.

The SET SCHEMA statement is not transactional: If the SET SCHEMA statement is part of a transaction that is rolled back, the schema change remains in effect.

Syntax

```
SET [CURRENT] SCHEMA [=]
{ schemaName|
USER | ? | '<string-constant>' } | SET CURRENT SQLID [=]
```

```
{
schemaName| USER | ? | '<string-constant>' }
```

schemaName is an identifier with a maximum length of 128. It is case insensitive unless enclosed in double quotes. (For example, SYS is equivalent to sYs, SYs, sys, etcetera.)

USER is the current user. If no current user is defined, the current schema defaults the *APP* schema. (If a user name was specified upon connection, the user's name is the default schema for the connection, if a schema with that name exists.)

? is a dynamic parameter specification that can be used in prepared statements. The SET SCHEMA statement can be prepared once and then executed with different schema values. The schema values are treated as string constants so they are case sensitive. For example, to designate the *APP* schema, use the string "APP" rather than "app".

```
-- the following are all equivalent and will work
-- assuming a schema called HOTEL
SET SCHEMA HOTEL
SET SCHEMA hotel
SET CURRENT SCHEMA hotel
SET CURRENT SQLID hotel
SET SCHEMA = hotel
SET CURRENT SCHEMA = hotel
SET CURRENT SQLID = hotel
SET SCHEMA "HOTEL" -- quoted identifier
SET SCHEMA 'HOTEL' -- quoted string--This example produces an error
 because
 --lower case hotel won't be found
SET SCHEMA = 'hotel'
 --This example produces an error because SQLID is not
 --allowed without CURRENT
SET SQLID hotel
 -- This sets the schema to the current user id
SET CURRENT SCHEMA USER
// Here's an example of using set schema in an Java program
PreparedStatement ps = conn.PrepareStatement("set schema ?");
ps.setString(1,"HOTEL");
ps.executeUpdate();
... do some work
ps.setString(1,"APP");
ps.executeUpdate();

ps.setString(1,"app"); //error - string is case sensitive
// no app will be found
ps.setNull(1, Types.VARCHAR); //error - null is not allowed
```

SELECT statement

Syntax

```
Query
[ORDER BY clause]
[result offset clause]
[fetch first clause]
[FOR UPDATE clause]
[WITH {RR|RS|CS|UR}]
```

A SELECT statement consists of a query with an optional ORDER BY clause, an optional result offset clause, an optional fetch first clause, an optional FOR UPDATE clause and optionally isolation level. The SELECT statement is so named because the typical first word of the query construct is SELECT. (*Query* includes the VALUES expression and UNION, INTERSECT, and EXCEPT expressions as well as SELECT expressions).

The ORDER BY clause guarantees the ordering of the *ResultSet*. The result offset clause and the fetch first clause can be used to fetch only a subset of the otherwise selected

rows, possibly with an offset into the result set. The FOR UPDATE clause makes the result set's cursor updatable. The SELECT statement supports the FOR FETCH ONLY clause. The FOR FETCH ONLY clause is synonymous with the FOR READ ONLY clause.

You can set the isolation level in a SELECT statement using the WITH {RR|RS|CS|UR} syntax.

Example

```
-- lists the names of the expression
-- SAL+BONUS+COMM as TOTAL_PAY and
-- orders by the new name TOTAL_PAY
SELECT FIRSTNME, SALARY+BONUS+COMM AS TOTAL_PAY
     FROM EMPLOYEE
     ORDER BY TOTAL_PAY
-- creating an updatable cursor with a FOR UPDATE clause
-- to update the start date (PRSTDATE) and the end date (PRENDATE)
-- columns in the PROJECT table
SELECT PROJNO, PRSTDATE, PRENDATE
      FROM PROJECT
      FOR UPDATE OF PRSTDATE, PRENDATE
-- set the isolation level to RR for this statement only
SELECT *
FROM Flights
WHERE flight_id BETWEEN 'AA1111' AND 'AA1112'
WITH RR
```

A SELECT statement returns a *ResultSet*. A *cursor* is a pointer to a specific row in *ResultSet*. In Java applications, all *ResultSets* have an underlying associated SQL cursor, often referred to as the result set's cursor. The cursor can be updatable, that is, you can update or delete rows as you step through the *ResultSet* if the SELECT statement that generated it and its underlying query meet cursor updatability requirements, as detailed below. The FOR UPDATE clause can be used to ensure a compilation check that the SELECT statement meets the requiremments of a updatable cursors, or to limit the columns that can be updated.
Note: The ORDER BY clause allows you to order the results of the SELECT. Without the ORDER BY clause, the results are returned in random order.

Requirements for updatable cursors and updatable ResultSets
Only simple, single-table SELECT cursors can be updatable. The SELECT statement for updatable ResultSets has the same syntax as the SELECT statement for updatable cursors. To generate updatable cursors:

- The SELECT statement must not include an ORDER BY clause.
- The underlying *Query* must be a *SelectExpression.*
- The *SelectExpression* in the underlying Query must not include:
  - DISTINCT
  - Aggregates
  - GROUP BY clause
  - HAVING clause
  - ORDER BY clause
- The FROM clause in the underlying *Query* must not have:
  - more than one table in its FROM clause
  - anything other than one table name
  - *SelectExpressions*
  - subqueries
- If the underlying *Query* has a WHERE clause, the WHERE clause must not have subqueries.

Note: Cursors are read-only by default. To produce an updatable cursor besides meeting the requirements listed above, the concurrency mode for the ResultSet must be

ResultSet.CONCUR_UPDATABLE or the SELECT statement must have FOR UPDATE in the FOR clause (see FOR UPDATE clause).

There is no SQL language statement to *assign* a name to a cursor. Instead, one can use the JDBC API to assign names to cursors or retrieve system-generated names. For more information, see Naming or accessing the name of a cursor in the *Java DB Developer's Guide*.

Statement dependency system

The SELECT depends on all the tables and views named in the query and the conglomerates (units of storage such as heaps and indexes) chosen for access paths on those tables. CREATE INDEX does not invalidate a prepared SELECT statement. A DROP INDEX statement invalidates a prepared SELECT statement if the index is an access path in the statement. If the SELECT includes views, it also depends on the dictionary objects on which the view itself depends (see CREATE VIEW statement).

Any prepared UPDATE WHERE CURRENT or DELETE WHERE CURRENT statement against a cursor of a SELECT depends on the SELECT. Removing a SELECT through a *java.sql.Statement.close* request invalidates the UPDATE WHERE CURRENT or DELETE WHERE CURRENT.

The SELECT depends on all aliases used in the query. Dropping an alias invalidates a prepared SELECT statement if the statement uses the alias.

UPDATE statement

Syntax

```
{
    UPDATE table-Name [[AS] correlation-Name]
        SET column-Name = Value
        [ , column-Name = Value} ]*
        [WHERE clause] |
    UPDATE table-Name
        SET column-Name = Value
        [ , column-Name = Value ]*
        WHERE CURRENT OF
}
```

where *Value* is defined as follows:

```
Expression | DEFAULT
```

The first syntactical form, called a searched update, updates the value of one or more columns for all rows of the table for which the WHERE clause evaluates to TRUE.

The second syntactical form, called a positioned update, updates one or more columns on the current row of an open, updatable cursor. If columns were specified in the FOR UPDATE clause of the SELECT statement used to generate the cursor, only those columns can be updated. If no columns were specified or the select statement did not include a FOR UPDATE clause, all columns may be updated.

Specifying DEFAULT for the update value sets the value of the column to the default defined for that table.

The DEFAULT literal is the only value which you can directly assign to a generated column. Whenever you alter the value of a column referenced by the *generation-clause* of a generated column, Derby recalculates the value of the generated column.

Example

```
-- All the employees except the manager of
-- department (WORKDEPT) 'E21' have been temporarily reassigned.
```

```
-- Indicate this by changing their job (JOB) to NULL and their pay
-- (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE WORKDEPT = 'E21' AND JOB <> 'MANAGER'

-- PROMOTE the job (JOB) of employees without a specific job title to
 MANAGER
UPDATE EMPLOYEE
 SET JOB = 'MANAGER'
 WHERE JOB IS NULL;
// Increase the project staffing (PRSTAFF) by 1.5 for all projects
stmt.executeUpdate("UPDATE PROJECT SET PRSTAFF = "
"PRSTAFF + 1.5" +
"WHERE CURRENT OF" + ResultSet.getCursorName());

-- Change the job (JOB) of employee number (EMPNO) '000290' in the
 EMPLOYEE table
-- to its DEFAULT value which is NULL
UPDATE EMPLOYEE
  SET JOB = DEFAULT
  WHERE EMPNO = '000290'
```

Statement dependency system

A searched update statement depends on the table being updated, all of its conglomerates (units of storage such as heaps or indexes), all of its constraints, and any other table named in the WHERE clause or SET expressions. A CREATE or DROP INDEX statement or an ALTER TABLE statement for the target table of a prepared searched update statement invalidates the prepared searched update statement.

The positioned update statement depends on the cursor and any tables the cursor references. You can compile a positioned update even if the cursor has not been opened yet. However, removing the open cursor with the JDBC *close* method invalidates the positioned update.

A CREATE or DROP INDEX statement or an ALTER TABLE statement for the target table of a prepared positioned update invalidates the prepared positioned update statement.

Dropping an alias invalidates a prepared update statement if the latter statement uses the alias.

Dropping or adding triggers on the target table of the update invalidates the update statement.

SQL clauses

CONSTRAINT clause

A CONSTRAINT clause is an optional part of a CREATE TABLE statement or ALTER TABLE statement. A constraint is a rule to which data must conform. Constraint names are optional.

A CONSTRAINT can be one of the following:

- a column-level constraint

  Column-level constraints refer to a single column in the table and do not specify a column name (except check constraints). They refer to the column that they follow.
- a table-level constraint

Table-level constraints refer to one or more columns in the table. Table-level constraints specify the names of the columns to which they apply. Table-level CHECK constraints can refer to 0 or more columns in the table.

Column constraints include:

- NOT NULL

  Specifies that this column cannot hold NULL values (constraints of this type are not nameable).
- PRIMARY KEY

  Specifies the column that uniquely identifies a row in the table. The identified columns must be defined as NOT NULL.

  Note: If you attempt to add a primary key using ALTER TABLE and any of the columns included in the primary key contain null values, an error will be generated and the primary key will not be added. See ALTER TABLE statement for more information.
- UNIQUE

  Specifies that values in the column must be unique.
- FOREIGN KEY

  Specifies that the values in the column must correspond to values in a referenced primary key or unique key column or that they are NULL.
- CHECK

  Specifies rules for values in the column.

Table constraints include:

- PRIMARY KEY

  Specifies the column or columns that uniquely identify a row in the table. NULL values are not allowed.
- UNIQUE

  Specifies that values in the columns must be unique.
- FOREIGN KEY

  Specifies that the values in the columns must correspond to values in referenced primary key or unique columns or that they are NULL.

  Note: If the foreign key consists of multiple columns, and *any* column is NULL, the whole key is considered NULL. The insert is permitted no matter what is on the non-null columns.
- CHECK

  Specifies a wide range of rules for values in the table.

Column constraints and table constraints have the same function; the difference is in where you specify them. Table constraints allow you to specify more than one column in a PRIMARY KEY, UNIQUE, CHECK, or FOREIGN KEY constraint definition. Column-level constraints (except for check constraints) refer to only one column.

Syntax

Primary key constraints

A primary key defines the set of columns that uniquely identifies rows in a table.

When you create a primary key constraint, none of the columns included in the primary key can have NULL constraints; that is, they must not permit NULL values.

ALTER TABLE ADD PRIMARY KEY allows you to include existing columns in a primary key if they were first defined as NOT NULL. NULL values are not allowed. If the

column(s) contain NULL values, the system will not add the primary key constraint. See ALTER TABLE statement for more information.

A table can have at most one PRIMARY KEY constraint.

Unique constraints

A UNIQUE constraint defines a set of columns that uniquely identify rows in a table only if all the key values are not NULL. If one or more key parts are NULL, duplicate keys are allowed.

For example, if there is a UNIQUE constraint on `col1` and `col2` of a table, the combination of the values held by `col1` and `col2` will be unique as long as these values are not NULL. If one of `col1` and `col2` holds a NULL value, there can be another identical row in the table.

A table can have multiple UNIQUE constraints.

Foreign key constraints

Foreign keys provide a way to enforce the referential integrity of a database. A foreign key is a column or group of columns within a table that references a key in some other table (or sometimes, though rarely, the same table). The foreign key must always include the columns of which the types exactly match those in the referenced primary key or unique constraint.

For a table-level foreign key constraint in which you specify the columns in the table that make up the constraint, you cannot use the same column more than once.

If there is a column list in the *ReferencesSpecification* (a list of columns in the referenced table), it must correspond either to a unique constraint or to a primary key constraint in the referenced table. The *ReferencesSpecification* can omit the column list for the referenced table if that table has a declared primary key.

If there is no column list in the *ReferencesSpecification* and the referenced table has no primary key, a statement exception is thrown. (This means that if the referenced table has only unique keys, you must include a column list in the *ReferencesSpecification.*)

A foreign key constraint is satisfied if there is a matching value in the referenced unique or primary key column. If the foreign key consists of multiple columns, the foreign key value is considered NULL if any of its columns contains a NULL.
Note: It is possible for a foreign key consisting of multiple columns to allow one of the columns to contain a value for which there is no matching value in the referenced columns, per the SQL-92 standard. To avoid this situation, create NOT NULL constraints on all of the foreign key's columns.

Foreign key constraints and DML

When you insert into or update a table with an enabled foreign key constraint, Derby checks that the row does not violate the foreign key constraint by looking up the corresponding referenced key in the referenced table. If the constraint is not satisfied, Derby rejects the insert or update with a statement exception.

When you update or delete a row in a table with a referenced key (a primary or unique constraint referenced by a foreign key), Derby checks every foreign key constraint that references the key to make sure that the removal or modification of the row does not cause a constraint violation. If removal or modification of the row would cause a constraint violation, the update or delete is not permitted and Derby throws a statement exception.

Derby performs constraint checks at the time the statement is executed, not when the transaction commits.

Backing indexes

UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints generate indexes that enforce or "back" the constraint (and are sometimes called *backing indexes*). PRIMARY KEY constraints generate unique indexes. FOREIGN KEY constraints generate non-unique indexes. UNIQUE constraints generate unique indexes if all the columns are non-nullable, and they generate non-unique indexes if one or more columns are nullable. Therefore, if a column or set of columns has a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint on it, you do not need to create an index on those columns for performance. Derby has already created it for you. See Indexes and constraints.

These indexes are available to the optimizer for query optimization (see CREATE INDEX statement) and have system-generated names.

You cannot drop backing indexes with a DROP INDEX statement; you must drop the constraint or the table.

Check constraints

A check constraint can be used to specify a wide range of rules for the contents of a table. A search condition (which is a boolean expression) is specified for a check constraint. This search condition must be satisfied for all rows in the table. The search condition is applied to each row that is modified on an INSERT or UPDATE at the time of the row modification. The entire statement is aborted if any check constraint is violated.

Requirements for search condition

If a check constraint is specified as part of a column-definition, a column reference can only be made to the same column. Check constraints specified as part of a table definition can have column references identifying columns previously defined in the CREATE TABLE statement.

The search condition must always return the same value if applied to the same values. Thus, it cannot contain any of the following:

- Dynamic parameters (?)
- Date/Time Functions (CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP)
- Subqueries
- User Functions (such as USER, SESSION_USER, CURRENT_USER)

Referential actions

You can specify an ON DELETE clause and/or an ON UPDATE clause, followed by the appropriate action (CASCADE, RESTRICT, SET NULL, or NO ACTION) when defining foreign keys. These clauses specify whether Derby should modify corresponding foreign key values or disallow the operation, to keep foreign key relationships intact when a primary key value is updated or deleted from a table.

You specify the update and delete rule of a referential constraint when you define the referential constraint.

The update rule applies when a row of either the parent or dependent table is updated. The choices are NO ACTION and RESTRICT.

When a value in a column of the parent table's primary key is updated and the update rule has been specified as RESTRICT, Derby checks dependent tables for foreign key constraints. If any row in a dependent table violates a foreign key constraint, the transaction is rolled back.

If the update rule is NO ACTION, Derby checks the dependent tables for foreign key constraints *after* all updates have been executed but *before* triggers have been executed. If any row in a dependent table violates a foreign key constraint, the statement is rejected.

When a value in a column of the dependent table is updated, and that value is part of a foreign key, NO ACTION is the implicit update rule. NO ACTION means that if a foreign key is updated with a non-null value, the update value must match a value in the parent table's primary key when the update statement is completed. If the update does not match a value in the parent table's primary key, the statement is rejected.

The delete rule applies when a row of the parent table is deleted and that row has dependents in the dependent table of the referential constraint. If rows of the dependent table are deleted, the delete operation on the parent table is said to be *propagated* to the dependent table. If the dependent table is also a parent table, the action specified applies, in turn, to its dependents.

The choices are NO ACTION, RESTRICT, CASCADE, or SET NULL. SET NULL can be specified only if some column of the foreign key allows null values.

If the delete rule is:

NO ACTION, Derby checks the dependent tables for foreign key constraints *after* all deletes have been executed but *before* triggers have been executed. If any row in a dependent table violates a foreign key constraint, the statement is rejected.

RESTRICT, Derby checks dependent tables for foreign key constraints. If any row in a dependent table violates a foreign key constraint, the transaction is rolled back.

CASCADE, the delete operation is propagated to the dependent table (and that table's dependents, if applicable).

SET NULL, each nullable column of the dependent table's foreign key is set to null. (Again, if the dependent table also has dependent tables, nullable columns in those tables' foreign keys are also set to null.)

Each referential constraint in which a table is a parent has its own delete rule; all applicable delete rules are used to determine the result of a delete operation. Thus, a row cannot be deleted if it has dependents in a referential constraint with a delete rule of RESTRICT or NO ACTION. Similarly, a row cannot be deleted if the deletion cascades to any of its descendants that are dependents in a referential constraint with the delete rule of RESTRICT or NO ACTION.

Deleting a row from the parent table involves other tables. Any table involved in a delete operation on the parent table is said to be delete-connected to the parent table. The delete can affect rows of these tables in the following ways:

- If the delete rule is RESTRICT or NO ACTION, a dependent table is involved in the operation but is not affected by the operation. (That is, Derby checks the values within the table, but does not delete any values.)
- If the delete rule is SET NULL, a dependent table's rows can be updated when a row of the parent table is the object of a delete or propagated delete operation.
- If the delete rule is CASCADE, a dependent table's rows can be deleted when a parent table is the object of a delete.
- If the dependent table is also a parent table, the actions described in this list apply, in turn, to its dependents.

Examples

```
-- column-level primary key constraint named OUT_TRAY_PK:
CREATE TABLE SAMP.OUT_TRAY
 (
 SENT TIMESTAMP,
 DESTINATION CHAR(8),
 SUBJECT CHAR(64) NOT NULL CONSTRAINT OUT_TRAY_PK PRIMARY KEY,
 NOTE_TEXT VARCHAR(3000)
   );

-- the table-level primary key definition allows you to
```

```
-- include two columns in the primary key definition:
CREATE TABLE SAMP.SCHED
 (
 CLASS_CODE CHAR(7) NOT NULL,
 DAY SMALLINT NOT NULL,
 STARTING TIME,
 ENDING TIME,
 PRIMARY KEY (CLASS_CODE, DAY)
 );

-- Use a column-level constraint for an arithmetic check
-- Use a table-level constraint
-- to make sure that a employee's taxes does not
-- exceed the bonus
CREATE TABLE SAMP.EMP
 (
 EMPNO CHAR(6) NOT NULL CONSTRAINT EMP_PK PRIMARY KEY,
 FIRSTNME CHAR(12) NOT NULL,
 MIDINIT vARCHAR(12) NOT NULL,
 LASTNAME VARCHAR(15) NOT NULL,
 SALARY DECIMAL(9,2) CONSTRAINT SAL_CK CHECK (SALARY >= 10000),
 BONUS DECIMAL(9,2),
 TAX DECIMAL(9,2),
 CONSTRAINT BONUS_CK CHECK (BONUS > TAX)
 );

-- use a check constraint to allow only appropriate
-- abbreviations for the meals
CREATE TABLE FLIGHTS
 (
 FLIGHT_ID CHAR(6) NOT NULL ,
 SEGMENT_NUMBER INTEGER NOT NULL ,
 ORIG_AIRPORT CHAR(3),
 DEPART_TIME TIME,
 DEST_AIRPORT CHAR(3),
 ARRIVE_TIME TIME,
 MEAL CHAR(1) CONSTRAINT MEAL_CONSTRAINT
 CHECK (MEAL IN ('B', 'L', 'D', 'S')),
 PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER)
 );

CREATE TABLE METROPOLITAN
 (
 HOTEL_ID INT NOT NULL CONSTRAINT HOTELS_PK PRIMARY KEY,
 HOTEL_NAME VARCHAR(40) NOT NULL,
 CITY_ID INT CONSTRAINT METRO_FK REFERENCES CITIES
 );

-- create a table with a table-level primary key constraint
-- and a table-level foreign key constraint
CREATE TABLE FLTAVAIL
 (
 FLIGHT_ID CHAR(6) NOT NULL,
 SEGMENT_NUMBER INT NOT NULL,
 FLIGHT_DATE DATE NOT NULL,
 ECONOMY_SEATS_TAKEN INT,
 BUSINESS_SEATS_TAKEN INT,
 FIRSTCLASS_SEATS_TAKEN INT,
 CONSTRAINT FLTAVAIL_PK PRIMARY KEY (FLIGHT_ID, SEGMENT_NUMBER),
 CONSTRAINT FLTS_FK
 FOREIGN KEY (FLIGHT_ID, SEGMENT_NUMBER)
 REFERENCES Flights (FLIGHT_ID, SEGMENT_NUMBER)
 );
-- add a unique constraint to a column
ALTER TABLE SAMP.PROJECT
ADD CONSTRAINT P_UC UNIQUE (PROJNAME);

-- create a table whose city_id column references the
-- primary key in the Cities table
-- using a column-level foreign key constraint
```

```
CREATE TABLE CONDOS
 (
 CONDO_ID INT NOT NULL CONSTRAINT hotels_PK PRIMARY KEY,
 CONDO_NAME VARCHAR(40) NOT NULL,
 CITY_ID INT CONSTRAINT city_foreign_key
 REFERENCES Cities ON DELETE CASCADE ON UPDATE RESTRICT
 );
```

Statement dependency system

INSERT and UPDATE statements depend on all constraints on the target table. DELETEs depend on unique, primary key, and foreign key constraints. These statements are invalidated if a constraint is added to or dropped from the target table.

Column-level-constraint

```
{
    NOT NULL |
    [ [CONSTRAINT constraint-Name]
    {
        CHECK (searchCondition) |
        {
            PRIMARY KEY |
            UNIQUE |
            REFERENCES clause
        }
    }
}
```

Table-level constraint

```
[CONSTRAINT constraint-Name]
{
    CHECK (searchCondition) |
    {
        PRIMARY KEY ( Simple-column-Name [ , Simple-column-Name ]* ) |
        UNIQUE ( Simple-column-Name [ , Simple-column-Name ]* ) |
        FOREIGN KEY ( Simple-column-Name
 [ , Simple-column-Name ]*
 )
REFERENCES clause
    }
}
```

References specification

```
REFERENCES table-Name [ ( Simple-column-Name [ , Simple-column-Name ]* )
 ]
[ ON DELETE {NO ACTION | RESTRICT | CASCADE | SET NULL}]
   [ ON UPDATE {NO ACTION | RESTRICT }]
|
[ ON UPDATE {NO ACTION | RESTRICT }] [ ON DELETE
   {NO ACTION | RESTRICT | CASCADE | SET NULL}]
```

searchCondition

A *searchCondition* is any Boolean expression that meets the requirements specified in Requirements for search condition.

If a *constraint-Name* is not specified, Derby generates a unique constraint name (for either column or table constraints).

FOR UPDATE clause

The FOR UPDATE clause is an optional part of a SELECT statement. Cursors are read-only by default. The FOR UPDATE clause specifies that the cursor should be updatable, and enforces a check during compilation that the SELECT statement meets

the requirements for an updatable *cursor*. For more information about updatability, see Requirements for updatable cursors and updatable ResultSets.

Syntax

```
FOR
{
    READ ONLY | FETCH ONLY |
    UPDATE [ OF Simple-column-Name [ , Simple-column-Name]* ]
}
```

Simple-column-Name refers to the names visible for the table specified in the FROM clause of the underlying query.

Note: The use of the FOR UPDATE clause is not mandatory to obtain an updatable JDBC ResultSet. As long as the statement used to generate the JDBC ResultSet meets the requirements for updatable cursor, it is sufficient for the JDBC Statement that generates the JDBC ResultSet to have concurrency mode `ResultSet.CONCUR_UPDATABLE` for the ResultSet to be updatable.

The optimizer is able to use an index even if the column in the index is being updated.

```
SELECT RECEIVED, SOURCE, SUBJECT, NOTE_TEXT FROM SAMP.IN_TRAY FOR UPDATE
```

For information about how indexes affect performance, see *Tuning Java DB*.

FROM clause

The FROM clause is a mandatory clause in a *SelectExpression.* It specifies the tables (*TableExpression*) from which the other clauses of the query can access columns for use in expressions.

Syntax

```
FROM TableExpression [ , TableExpression ] *
```

```
SELECT Cities.city_id
FROM Cities
WHERE city_id < 5
-- other types of TableExpressions
SELECT TABLENAME, ISINDEX
FROM SYS.SYSTABLES T, SYS.SYSCONGLOMERATES C
WHERE T.TABLEID = C.TABLEID
ORDER BY TABLENAME, ISINDEX
-- force the join order
SELECT *
FROM Flights, FlightAvailability
WHERE FlightAvailability.flight_id = Flights.flight_id
AND FlightAvailability.segment_number = Flights.segment_number
AND Flights.flight_id < 'AA1115'
-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME, FLIGHTS.DEST_AIRPORT
FROM COUNTRIES LEFT OUTER JOIN CITIES
ON COUNTRIES.COUNTRY_ISO_CODE = CITIES.COUNTRY_ISO_CODE
LEFT OUTER JOIN FLIGHTS
ON Cities.AIRPORT = FLIGHTS.DEST_AIRPORT
```

GROUP BY clause

A GROUP BY clause, part of a *SelectExpression*, groups a result into subsets that have matching values for one or more columns. In each group, no two rows have the same value for the grouping column or columns. NULLs are considered equivalent for grouping purposes.

You typically use a GROUP BY clause in conjunction with an aggregate expression.

Syntax

```
GROUP BY column-Name [ , column-Name ] *
```

column-Name must be a column from the current scope of the query; there can be no columns from a query block outside the current scope. For example, if a GROUP BY clause is in a subquery, it cannot refer to columns in the outer query.

SelectItems in the *SelectExpression* with a GROUP BY clause must contain only aggregates or grouping columns.

```
-- find the average flying_times of flights grouped by
-- airport
SELECT AVG (flying_time), orig_airport
FROM Flights
GROUP BY orig_airport

SELECT MAX(city_name), region
FROM Cities, Countries
WHERE Cities.country_ISO_code = Countries.country_ISO_code
GROUP BY region
-- group by an a smallint
SELECT ID, AVG(SALARY)
FROM SAMP.STAFF
GROUP BY ID
-- Get the AVGSALARY and EMPCOUNT columns, and the DEPTNO column using
 the AS clause
-- And group by the WORKDEPT column using the correlation name OTHERS
SELECT OTHERS.WORKDEPT AS DEPTNO,
AVG(OTHERS.SALARY) AS AVGSALARY,
COUNT(*) AS EMPCOUNT
FROM SAMP.EMPLOYEE OTHERS
GROUP BY OTHERS.WORKDEPT
```

HAVING clause

A HAVING clause restricts the results of a GROUP BY in a *SelectExpression.* The HAVING clause is applied to each group of the grouped table, much as a WHERE clause is applied to a select list. If there is no GROUP BY clause, the HAVING clause is applied to the entire result as a single group. The SELECT clause cannot refer directly to any column that does not have a GROUP BY clause. It can, however, refer to constants, aggregates, and special registers.

Syntax

```
HAVING searchCondition
```

The *searchCondition*, which is a specialized *booleanExpression*, can contain only grouping columns (see GROUP BY clause), columns that are part of aggregate expressions, and columns that are part of a subquery. For example, the following query is illegal, because the column SALARY is not a grouping column, it does not appear within an aggregate, and it is not within a subquery:

```
-- SELECT COUNT(*)
-- FROM SAMP.STAFF
-- GROUP BY ID
-- HAVING SALARY > 15000
```

Aggregates in the HAVING clause do not need to appear in the SELECT list. If the HAVING clause contains a subquery, the subquery can refer to the outer query block if and only if it refers to a grouping column.

```
-- Find the total number of economy seats taken on a flight,
-- grouped by airline,
-- only when the group has at least 2 records.
SELECT SUM(ECONOMY_SEATS_TAKEN), AIRLINE_FULL
FROM FLIGHTAVAILABILITY, AIRLINES
WHERE SUBSTR(FLIGHTAVAILABILITY.FLIGHT_ID, 1, 2) = AIRLINE
GROUP BY AIRLINE_FULL
HAVING COUNT(*) > 1
```

ORDER BY clause

The ORDER BY clause is an optional element of a SELECT statement. An ORDER BY clause allows you to specify the order in which rows appear in the *ResultSet*.

Syntax

```
ORDER BY { column-Name | ColumnPosition | Expression }
    [ ASC | DESC ]
    [ , column-Name | ColumnPosition | Expression
        [ ASC | DESC ] ] *
```

column-Name
: Refers to the names visible from the *SelectItems* in the underlying query of the SELECT statement. The column-Name that you specify in the ORDER BY clause does not need to be the SELECT list.

ColumnPosition
: An integer that identifies the number of the column in the *SelectItems* in the underlying query of the SELECT statement. ColumnPosition must be greater than 0 and not greater than the number of columns in the result table. In other words, if you want to order by a column, that column must be specified in the SELECT list.

Expression
: A sort key expression, such as numeric, string, and datetime expressions. *Expression* can also be a row value expression such as a scalar subquery or case expression.

ASC
: Specifies that the results should be returned in ascending order. If the order is not specified, ASC is the default.

DESC
: Specifies that the results should be returned in descending order.

Notes

- If SELECT DISTINCT is specified or if the SELECT statement contains a GROUP BY clause, the ORDER BY columns must be in the SELECT list.
- An ORDER BY clause prevents a SELECT statement from being an updatable cursor. For more information, see Requirements for updatable cursors and updatable ResultSets. For example, if an INTEGER column contains integers, NULL is considered greater than 1 for purposes of sorting. In other words, NULL values are sorted high.

Example using a correlation name

You can sort the result set by a correlation name, if the correlation name is specified in the select list. For example, to return from the CITIES database all of the entries in the CITY_NAME and COUNTRY columns, where the COUNTRY column has the correlation name NATION, you specify this SELECT statement:

```
SELECT CITY_NAME, COUNTRY AS NATION
    FROM CITIES
    ORDER BY NATION
```

Example using a numeric expression

You can sort the result set by a numeric expression, for example:

```
SELECT name, salary, bonus FROM employee
   ORDER BY salary+bonus
```

In this example, the salary and bonus columns are DECIMAL data types.

Example using a function
You can sort the result set by invoking a function, for example:

```
SELECT i, len FROM measures
   ORDER BY sin(i)
```

The result offset and fetch first clauses

The *result offset clause* provides a way to skip the N first rows in a result set before starting to return any rows. The *fetch first clause*, which can be combined with the *result offset clause* if desired, limits the number of rows returned in the result set. The *fetch first clause* can sometimes be useful for retrieving only a few rows from an otherwise large result set, usually in combination with an ORDER BY clause. The use of this clause can give efficiency benefits. In addition, it can make programming the application simpler.

Syntax

```
OFFSET integer-literal {ROW | ROWS}
```

```
FETCH { FIRST | NEXT } [integer-literal] {ROW | ROWS} ONLY
```

ROW is synonymous with ROWS and FIRST is synonymous with NEXT.

For the *result offset clause* the integer literal must be equal to 0 (default if the clause is not given), or positive. If it is larger than the number of rows in the underlying result set, no rows are returned.

For the *fetch first clause*, the literal must be 1 or higher. The literal can be omitted, in which case it defaults to 1. If the clause is omitted entirely, all rows (or those rows remaining if a *result offset clause* is also given) will be returned.

Examples

```
-- Fetch the first row of T
SELECT * FROM T FETCH FIRST ROW ONLY

-- Sort T using column I, then fetch rows 11 through 20 of the sorted
--   rows (inclusive)
SELECT * FROM T ORDER BY I OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY

-- Skip the first 100 rows of T
-- If the table has fewer than 101 records, an empty result set is
--   returned
SELECT * FROM T OFFSET 100 ROWS
```

Note: Make sure to specify the ORDER BY clause if you expect to retrieve a sorted result set. If you do not use an ORDER BY clause, the result set that is retrieved will typically have the order in which the records were inserted.

WHERE clause

A WHERE clause is an optional part of a *SelectExpression*, DELETE statement, or UPDATE statement. The WHERE clause lets you select rows based on a boolean expression. Only rows for which the expression evaluates to TRUE are returned in the result, or, in the case of a DELETE statement, deleted, or, in the case of an UPDATE statement, updated.

Syntax

```
WHERE Boolean expression
```

Boolean expressions are allowed in the WHERE clause. Most of the general expressions listed in Table of general expressions, can result in a boolean value.

In addition, there are the more common boolean expressions. Specific boolean operators listed in Table 10, take one or more operands; the expressions return a boolean value.

Example

```
-- find the flights where no business-class seats have
-- been booked
SELECT *
FROM FlightAvailability
WHERE business_seats_taken IS NULL
OR business_seats_taken = 0
-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result.
SELECT SAMP.EMP_ACT.*, LASTNAME
  FROM SAMP.EMP_ACT, SAMP.EMPLOYEE
  WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
-- Determine the employee number and salary of sales representatives
-- along with the average salary and head count of their departments.
-- This query must first create a new-column-name specified in the AS
 clause
-- which is outside the fullselect (DINFO)
-- in order to get the AVGSALARY and EMPCOUNT columns,
-- as well as the DEPTNO column that is used in the WHERE clause
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
 FROM EMPLOYEE THIS_EMP,
   (SELECT OTHERS.WORKDEPT AS DEPTNO,
           AVG(OTHERS.SALARY) AS AVGSALARY,
           COUNT(*) AS EMPCOUNT
    FROM EMPLOYEE OTHERS
    GROUP BY OTHERS.WORKDEPT
   )AS DINFO
 WHERE THIS_EMP.JOB = 'SALESREP'
    AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

WHERE CURRENT OF clause

The WHERE CURRENT OF clause is a clause in some UPDATE and DELETE statements. It allows you to perform positioned updates and deletes on updatable cursors. For more information about updatable cursors, see SELECT statement.

Syntax

```
WHERE CURRENT OF cursor-Name
```

```
Statement s = conn.createStatement();
s.setCursorName("AirlinesResults");
ResultSet rs = conn.executeQuery(
    "SELECT Airline, basic_rate " +
    "FROM Airlines FOR UPDATE OF basic_rate");
Statement s2 = conn.createStatement();
s2.executeUpdate("UPDATE Airlines SET basic_rate = basic_rate " +
    "+ .25 WHERE CURRENT OF AirlinesResults");
```

SQL expressions

Syntax for many statements and expressions includes the term Expression, or a term for a specific kind of expression such as *TableSubquery*. Expressions are allowed in these specified places within statements.

Some locations allow only a specific type of expression or one with a specific property. If not otherwise specified, an expression is permitted anywhere the word *Expression* appears in the syntax. This includes:

- ORDER BY clause
- *SelectExpression*
- UPDATE statement (SET portion)
- VALUES Expression
- WHERE clause

Of course, many other statements include these elements as building blocks, and so allow expressions as part of these elements.

The following tables list all the possible SQL expressions and indicate where the expressions are allowed.

General expressions

General expressions are expressions that might result in a value of any type.

Table 2. Table of general expressions

Expression Type	Explanation
Column reference	A *column-Name* that references the value of the column made visible to the expression containing the Column reference. You must qualify the *column-Name* by the table name or correlation name if it is ambiguous. The qualifier of a *column-Name* must be the correlation name, if a correlation name is given to a table that is in a FROM clause. The table name is no longer visible as a *column-Name* qualifier once it has been aliased by a correlation name. Allowed in *SelectExpressions*, UPDATE statements, and the WHERE clauses of data manipulation statements.
Constant	Most built-in data types typically have constants associated with them (as shown in Data types).
NULL	NULL is an untyped constant representing the unknown value. Allowed in CAST expressions or in INSERT VALUES lists and UPDATE SET clauses. Using it in a CAST expression gives it a specific data type.
Dynamic parameter	A dynamic parameter is a parameter to an SQL statement for which the value is not specified when the statement is created. Instead, the statement has a question mark (?) as a placeholder for each dynamic parameter. See Dynamic parameters. Dynamic parameters are permitted only in prepared statements. You must specify values for them before the prepared statement is executed. The values specified must match the types expected.

Expression Type	Explanation
	Allowed anywhere in an expression where the data type can be easily deduced. See Dynamic parameters.
CAST expression	Lets you specify the type of NULL or of a dynamic parameter or convert a value to another type. See CAST function.
Scalar subquery	Subquery that returns a single row with a single column. See *ScalarSubquery.*
Table subquery	Subquery that returns more than one column and more than one row. See *TableSubquery.* Allowed as a tableExpression in a FROM clause and with EXISTS, IN, and quantified comparisons.
Conditional expression	A conditional expression chooses an expression to evaluate based on a boolean test.

Boolean expressions

Boolean expressions are expressions that result in boolean values. Most general expressions can result in boolean values. Boolean expressions commonly used in a WHERE clause are made of operands operated on by SQL operators. See SQL Boolean Operators.

Numeric expressions

Numeric expressions are expressions that result in numeric values. Most of the general expressions can result in numeric values. Numeric values have one of the following types:

- BIGINT
- DECIMAL
- DOUBLE PRECISION
- INTEGER
- REAL
- SMALLINT

Table 3. Table of numeric expressions

Expression Type	Explanation
+, -, *, /, unary + and - expressions	Evaluate the expected math operation on the operands. If both operands are the same type, the result type is not promoted, so the division operator on integers results in an integer that is the truncation of the actual numeric result. When types are mixed, they are promoted as described in Data types. Unary + is a noop (i.e., +4 is the same as 4). Unary - is the same as multiplying the value by -1, effectively changing its sign.
AVG	Returns the average of a set of numeric values. AVG function
SUM	Returns the sum of a set of numeric values. SUM function

Expression Type	Explanation
LENGTH	Returns the number of characters in a character or bit string. See LENGTH function.
LOWER	See LCASE or LOWER function.
COUNT	Returns the count of a set of values. See COUNT function, COUNT(*) function.

Character expressions
Character expressions are expressions that result in a CHAR or VARCHAR value. Most general expressions can result in a CHAR or VARCHAR value.
Table 4. Table of character expressions

Expression Type	Explanation
A CHAR or VARCHAR value that uses wildcards.	The wildcards % and _ make a character string a pattern against which the LIKE operator can look for a match.
Concatenation expression	In a concatenation expression, the concatenation operator, "||", concatenates its right operand to the end of its left operand. Operates on character and bit strings. See Concatenation operator.
Built-in string functions	The built-in string functions act on a String and return a string. See LTRIM function, LCASE or LOWER function, RTRIM function, TRIM function, SUBSTR function, and UCASE or UPPER function.
USER functions	User functions return information about the current user as a String. See CURRENT_USER function, SESSION_USER function, and .

Date and time expressions
A date or time expression results in a DATE, TIME, or TIMESTAMP value. Most of the general expressions can result in a date or time value.
Table 5. Table of date and time expressions

Expression type	Explanation
CURRENT_DATE	Returns the current date. See CURRENT_DATE function.
CURRENT_TIME	Returns the current time. See CURRENT_TIME function.
CURRENT_TIMESTAMP	Returns the current timestamp. See CURRENT_TIMESTAMP function.

SelectExpression

A *SelectExpression* is the basic SELECT-FROM-WHERE construct used to build a table value based on filtering and projecting values from other tables.

Syntax

```
SELECT [ DISTINCT | ALL ] SelectItem [
 , SelectItem
 ]*
FROM clause
[ WHERE clause]
[ GROUP BY clause ]
[ HAVING clause ]
```

SelectItem:

```
{
    * |
    { table-Name | correlation-Name } .* |
    Expression [AS Simple-column-Name]
}
```

The SELECT clause contains a list of expressions and an optional quantifier that is applied to the results of the FROM clause and the WHERE clause. If DISTINCT is specified, only one copy of any row value is included in the result. Nulls are considered duplicates of one another for the purposes of DISTINCT. If no quantifier, or ALL, is specified, no rows are removed from the result in applying the SELECT clause (ALL is the default).

A *SelectItem* projects one or more result column values for a table result being constructed in a *SelectExpression*.

The result of the FROM clause is the cross product of the FROM items. The WHERE clause can further qualify this result.

The WHERE clause causes rows to be filtered from the result based on a boolean expression. Only rows for which the expression evaluates to TRUE are returned in the result.

The GROUP BY clause groups rows in the result into subsets that have matching values for one or more columns. GROUP BY clauses are typically used with aggregates.

If there is a GROUP BY clause, the SELECT clause must contain *only* aggregates or grouping columns. If you want to include a non-grouped column in the SELECT clause, include the column in an aggregate expression. For example:

```
-- List head count of each department,
-- the department number (WORKDEPT), and the average departmental salary
 (SALARY)
-- for all departments in the EMPLOYEE table.
-- Arrange the result table in ascending order by average departmental
 salary.
SELECT COUNT(*),WORK_DEPT,AVG(SALARY)
     FROM EMPLOYEE
     GROUP BY WORK_DEPT
     ORDER BY 3
```

If there is no GROUP BY clause, but a *SelectItem* contains an aggregate not in a subquery, the query is implicitly grouped. The entire table is the single group.

The HAVING clause restricts a grouped table, specifying a search condition (much like a WHERE clause) that can refer only to grouping columns or aggregates from the current scope. The HAVING clause is applied to each group of the grouped table. If the HAVING clause evaluates to TRUE, the row is retained for further processing. If the HAVING clause evaluates to FALSE or NULL, the row is discarded. If there is a HAVING clause but no GROUP BY, the table is implicitly grouped into one group for the entire table.

Derby processes a *SelectExpression* in the following order:

- FROM clause
- WHERE clause
- GROUP BY (or implicit GROUP BY)

- HAVING clause
- SELECT clause

The result of a *SelectExpression* is always a table.

When a query does not have a FROM clause (when you are constructing a value, not getting data out of a table), you use a VALUES expression, not a *SelectExpression*. For example:

```
VALUES CURRENT_TIMESTAMP
```

See VALUES expression.

The * wildcard

* expands to all columns in the tables in the associated FROM clause.

*table-Name.** and *correlation-Name.** expand to all columns in the identified table. That table must be listed in the associated FROM clause.

Naming columns

You can name a *SelectItem* column using the AS clause. If a column of a *SelectItem* is not a simple *ColumnReference* expression or named with an AS clause, it is given a generated unique name.

These column names are useful in several cases:

- They are made available on the JDBC *ResultSetMetaData*.
- They are used as the names of the columns in the resulting table when the *SelectExpression* is used as a table subquery in a FROM clause.
- They are used in the ORDER BY clause as the column names available for sorting.

```
-- this example shows SELECT-FROM-WHERE
-- with an ORDER BY clause
-- and correlation-Names for the tables
SELECT CONSTRAINTNAME, COLUMNNAME
FROM SYS.SYSTABLES t, SYS.SYSCOLUMNS col,
SYS.SYSCONSTRAINTS cons, SYS.SYSCHECKS checks
WHERE t.TABLENAME = 'FLIGHTS' AND t.TABLEID = col.
REFERENCEID AND t.TABLEID = cons.TABLEID
AND cons.CONSTRAINTID = checks.CONSTRAINTID
ORDER BY CONSTRAINTNAME
-- This example shows the use of the DISTINCT clause
SELECT DISTINCT ACTNO
FROM EMP_ACT
-- This example shows how to rename an expression
-- Using the EMPLOYEE table, list the department number (WORKDEPT) and
-- maximum departmental salary (SALARY) renamed as BOSS
-- for all departments whose maximum salary is less than the
-- average salary in all other departments.
SELECT WORKDEPT AS DPT, MAX(SALARY) AS BOSS
 FROM EMPLOYEE EMP_COR
 GROUP BY WORKDEPT
 HAVING MAX(SALARY) < (SELECT AVG(SALARY)
    FROM EMPLOYEE
    WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
 ORDER BY BOSS
```

TableExpression

A *TableExpression* specifies a table, view, or function in a FROM clause. It is the source from which a *SelectExpression* selects a result.

A correlation name can be applied to a table in a *TableExpression* so that its columns can be qualified with that name. If you do not supply a correlation name, the table name qualifies the column name. When you give a table a correlation name, you cannot use

the table name to qualify columns. You must use the correlation name when qualifying column names.

No two items in the FROM clause can have the same correlation name, and no correlation name can be the same as an unqualified table name specified in that FROM clause.

In addition, you can give the columns of the table new names in the AS clause. Some situations in which this is useful:

- When a VALUES expression is used as a *TableSubquery,* since there is no other way to name the columns of a VALUES expression.
- When column names would otherwise be the same as those of columns in other tables; renaming them means you don't have to qualify them.

The Query in a *TableSubquery* appearing in a *FromItem* can contain multiple columns and return multiple rows. See *TableSubquery.*

For information about the optimizer overrides you can specify, see *Tuning Java DB*.

Syntax

```
{
TableViewOrFunctionExpression | JOIN operation
}
```

Example

```
-- SELECT from a Join expression
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
 FROM EMPLOYEE E LEFT OUTER JOIN
    DEPARTMENT INNER JOIN EMPLOYEE M
  ON MGRNO = M.EMPNO
  ON E.WORKDEPT = DEPTNO
```

TableViewOrFunctionExpression

```
{table-Name |view-Name | TableFunctionInvocation}
    [ [ AS ] correlation-Name
        [ (Simple-column-Name [ , Simple-column-Name]* ) ] ] ]
```

TableFunctionInvocation:

```
TABLE function-name( [ [ function-arg ] [, function-arg ]* ] )
```

Note that when you invoke a table function, you must bind it to a correlation name. For example:

```
SELECT s.*
FROM TABLE( externalEmployees( 42 ) ) s
```

VALUES expression

The VALUES expression allows construction of a row or a table from other values. A VALUES expression can be used in all the places where a query can, and thus can be used in any of the following ways:

- As a statement that returns a *ResultSet*
- Within expressions and statements wherever subqueries are permitted
- As the source of values for an INSERT statement (in an INSERT statement, you normally use a VALUES expression when you do not use a *SelectExpression*)

Syntax

```
{
    VALUES ( Value {, Value }* )
        [ , ( Value {, Value }* ) ]* |
    VALUES Value [ , Value ]*
 }
```

where *Value* is defined as

```
Expression | DEFAULT
```

The first form constructs multi-column rows. The second form constructs single-column rows, each expression being the value of the column of the row.

The DEFAULT keyword is allowed only if the VALUES expression is in an INSERT statement. Specifying DEFAULT for a column inserts the column's default value into the column. Another way to insert the default value into the column is to omit the column from the column list and only insert values into other columns in the table.

Examples

```
-- 3 rows of 1 column
VALUES (1),(2),(3)
-- 3 rows of 1 column
VALUES 1, 2, 3
-- 1 row of 3 columns
VALUES (1, 2, 3)
-- 3 rows of 2 columns
VALUES (1,21),(2,22),(3,23)
-- constructing a derived table
VALUES ('orange', 'orange'), ('apple', 'red'),
('banana', 'yellow')
-- Insert two new departments using one statement into the DEPARTMENT
 table,
-- but do not assign a manager to the new department.
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('B11', 'PURCHASING', 'B01'),
    ('E41', 'DATABASE ADMINISTRATION', 'E01')
-- insert a row with a DEFAULT value for the MAJPROJ column
INSERT INTO PROJECT (PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE,
 MAJPROJ)
VALUES ('PL2101', 'ENSURE COMPAT PLAN', 'B01', '000020', CURRENT_DATE,
 DEFAULT)

-- using a built-in function
VALUES CURRENT_DATE
-- getting the value of an arbitrary expression
VALUES (3*29, 26.0E0/3)
-- getting a value returned by a built-in function
values char(1)
```

Expression precedence

Precedence of operations from highest to lowest is:

- (), ?, Constant (including sign), NULL, *ColumnReference*, *ScalarSubquery*, CAST
- LENGTH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, and other built-ins
- unary + and -
- *, /, || (concatenation)
- binary + and -
- comparisons, quantified comparisons, EXISTS, IN, IS NULL, LIKE, BETWEEN, IS
- NOT
- AND
- OR

You can explicitly specify precedence by placing expressions within parentheses. An expression within parentheses is evaluated before any operations outside the parentheses are applied to it.

Example

```
(3+4)*9
(age < 16 OR age > 65) AND employed = TRUE
```

Boolean expressions

Boolean expressions are allowed in WHERE clauses and in check constraints. Boolean expressions in check constraints have limitations not noted here; see CONSTRAINT clause for more information. Boolean expressions in a WHERE clause have a highly liberal syntax; see WHERE clause, for example.

A boolean expression can include a boolean operator or operators. These are listed in SQL Boolean Operators.

Table 6. SQL Boolean Operators

<table>
<tr><th>Operator</th><th>Explanation and Example</th><th>Syntax</th></tr>
<tr><td>AND, OR, NOT</td><td>Evaluate any operand(s) that are boolean expressions

<code>(orig_airport = 'SFO') OR
    (dest_airport = 'GRU')
    -- returns true</code></td><td><code>{
  Expression AND
    Expression |
  Expression OR
    Expression |
  NOT Expression
}</code></td></tr>
<tr><td>Comparisons</td><td><, =, >, <=, >=, <> are applicable to all of the built-in types.

<code>DATE('1998-02-26') <
    DATE('1998-03-01')
-- returns true</code></td><td><code>Expression
{
  < |
  = |
  > |
  <= |
  >= |
  <>
}
Expression</code></td></tr>
<tr><td>IS NULL, IS NOT NULL</td><td>Test whether the result of an expression is null or not.

<code>WHERE MiddleName IS NULL</code></td><td><code>Expression IS [NOT
 ]
  NULL</code></td></tr>
<tr><td>LIKE</td><td>Attempts to match a character expression to a character pattern, which is a character string that includes one or more wildcards.

% matches any number (zero or more) of characters in the corresponding position in first character expression.

_ matches one character in the corresponding position in the character expression.

Any other character matches only that character in the corresponding position in the character expression.

<code>city LIKE 'Sant_'</code></td><td><code>CharacterExpression
  [NOT] LIKE

 CharacterExpression
  WithWildCard
  [ESCAPE

'escapeCharacter']</code></td></tr>
</table>

Operator	Explanation and Example	Syntax
	To treat % or _ as constant characters, escape the character with an optional escape character, which you specify with the ESCAPE clause. `SELECT a FROM tabA WHERE a LIKE '%=_' ESCAPE '='` **Note:** When LIKE comparisons are used,Derby compares one character at a time for non-metacharacters. This is different than the way Derby processes = comparisons. The comparisons with the = operator compare the entire character string on left side of the = operator with the entire character string on the right side of the = operator. For more information, see **Character-based collation in Derby** in the *Java DB Developer's Guide*.	
BETWEEN	Tests whether the first operand is between the second and third operands. The second operand must be less than the third operand. Applicable only to types to which <= and >= can be applied. `WHERE booking_date BETWEEN` `    DATE('1998-02-26') AND` `    DATE('1998-03-01')`	`Expression [ NOT ]` `BETWEEN Expression` `  AND Expression`
IN	Operates on table subquery or list of values. Returns TRUE if the left expression's value is in the result of the table subquery or in the list of values. Table subquery can return multiple rows but must return a single column. `WHERE booking_date NOT IN` `    (SELECT booking_date FROM` `HotelBookings WHERE` ` rooms_available = 0)`	`{` `  Expression [ NOT ]` `    IN` `    TableSubquery` ` |` `  Expression [ NOT ]` `    IN ( Expression` `    [, Expression ]*` ` )` `}`
EXISTS	Operates on a table subquery. Returns TRUE if the table subquery returns any rows, and FALSE if it returns no rows. Table subquery can return multiple columns (only if you use * to denote multiple columns) and rows. `WHERE EXISTS` `  (SELECT *` `  FROM Flights` `  WHERE dest_airport = 'SFO'` `  AND orig_airport = 'GRU')`	`[NOT] EXISTS` ` TableSubquery`

<table>
<tr><th>Operator</th><th>Explanation and Example</th><th>Syntax</th></tr>
<tr><td>Quantified comparison</td><td>A quantified comparison is a comparison operator (<, =, >, <=, >=, <>) with ALL or ANY or SOME applied.

Operates on table subqueries, which can return multiple rows but must return a single column.

If ALL is used, the comparison must be true for all values returned by the table subquery. If ANY or SOME is used, the comparison must be true for at least one value of the table subquery. ANY and SOME are equivalent.

<code>WHERE normal_rate < ALL
(SELECT budget/550 FROM Groups)</code></td><td><code>Expression
  ComparisonOperator
  {
    ALL |
    ANY |
    SOME
  }
  TableSubquery</code></td></tr>
</table>

Dynamic parameters

You can prepare statements that are allowed to have parameters for which the value is not specified when the statement is prepared using *PreparedStatement* methods in the JDBC API. These parameters are called dynamic parameters and are represented by a ?.

The JDBC API documents refer to dynamic parameters as IN, INOUT, or OUT parameters. In SQL, they are always IN parameters.

New: Derby supports the interface *ParameterMetaData*, new in JDBC 3.0. This interface describes the number, type, and properties of prepared statement parameters. See the *Java DB Developer's Guide* for more information.

You must specify values for them before executing the statement. The values specified must match the types expected.

Dynamic parameters example

```
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE HotelAvailability SET rooms_available = " +
    "(rooms_available - ?) WHERE hotel_id = ? " +
    "AND booking_date BETWEEN ? AND ?");
-- this sample code sets the values of dynamic parameters
-- to be the values of program variables
ps2.setInt(1, numberRooms);
ps2.setInt(2, theHotel.hotelId);
ps2.setDate(3, arrival);
ps2.setDate(4, departure);
updateCount = ps2.executeUpdate();
```

Where dynamic parameters are allowed

You can use dynamic parameters anywhere in an expression where their data type can be easily deduced.

1. Use as the first operand of BETWEEN is allowed if one of the second and third operands is not also a dynamic parameter. The type of the first operand is assumed to be the type of the non-dynamic parameter, or the union result of their types if both are not dynamic parameters.

```
WHERE ? BETWEEN DATE('1996-01-01') AND ?
-- types assumed to be DATE
```

2. Use as the second or third operand of BETWEEN is allowed. Type is assumed to be the type of the left operand.

```
WHERE DATE('1996-01-01') BETWEEN ? AND ?
-- types assumed to be DATE
```

3. Use as the left operand of an IN list is allowed if at least one item in the list is not itself a dynamic parameter. Type for the left operand is assumed to be the union result of the types of the non-dynamic parameters in the list.

```
WHERE ? NOT IN (?, ?, 'Santiago')
-- types assumed to be CHAR
```

4. Use in the values list in an IN predicate is allowed if the first operand is not a dynamic parameter or its type was determined in the previous rule. Type of the dynamic parameters appearing in the values list is assumed to be the type of the left operand.

```
WHERE FloatColumn IN (?, ?,
 ?)
-- types assumed to be FLOAT
```

5. For the binary operators +, -, *, /, AND, OR, <, >, =, <>, <=, and >=, use of a dynamic parameter as one operand but not both is permitted. Its type is taken from the other side.

```
WHERE ? < CURRENT_TIMESTAMP
-- type assumed to be a TIMESTAMP
```

6. Use in a CAST is always permitted. This gives the dynamic parameter a type.

```
CALL valueOf(CAST (? AS VARCHAR(10)))
```

7. Use on either or both sides of LIKE operator is permitted. When used on the left, the type of the dynamic parameter is set to the type of the right operand, but with the maximum allowed length for the type. When used on the right, the type is assumed to be of the same length and type as the left operand. (LIKE is permitted on CHAR and VARCHAR types; see Concatenation operator for more information.)

```
WHERE ? LIKE 'Santi%'
-- type assumed to be CHAR with a length of
-- java.lang.Integer.MAX_VALUE
```

8. A ? parameter is allowed by itself on only one side of the || operator. That is, "? || ?" is not allowed. The type of a ? parameter on one side of a || operator is determined by the type of the expression on the other side of the || operator. If the expression on the other side is a CHAR or VARCHAR, the type of the parameter is VARCHAR with the maximum allowed length for the type. If the expression on the other side is a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA type, the type of the parameter is VARCHAR FOR BIT DATA with the maximum allowed length for the type.

```
SELECT BITcolumn || ?
FROM UserTable
-- Type assumed to be CHAR FOR BIT DATA of length specified for
 BITcolumn
```

9. In a conditional expression, which uses a ?, use of a dynamic parameter (which is also represented as a ?) is allowed. The type of a dynamic parameter as the first operand is assumed to be boolean. Only one of the second and third operands can be a dynamic parameter, and its type will be assumed to be the same as that of the other (that is, the third and second operand, respectively).

```
SELECT c1 IS NULL ? ? : c1
-- allows you to specify a "default" value at execution time
```

```
-- dynamic parameter assumed to be the type of c1
-- you cannot have dynamic parameters on both sides
-- of the :
```

10. A dynamic parameter is allowed as an item in the values list or select list of an INSERT statement. The type of the dynamic parameter is assumed to be the type of the target column.

```
INSERT INTO t VALUES (?)
-- dynamic parameter assumed to be the type
-- of the only column in table t
INSERT INTO t SELECT ?
FROM t2
-- not allowed
```

11. A ? parameter in a comparison with a subquery takes its type from the expression being selected by the subquery. For example:

```
SELECT *
FROM tab1
WHERE ? = (SELECT x FROM tab2)

SELECT *
FROM tab1
WHERE ? = ANY (SELECT x FROM tab2)
-- In both cases, the type of the dynamic parameter is
-- assumed to be the same as the type of tab2.x.
```

12. A dynamic parameter is allowed as the value in an UPDATE statement. The type of the dynamic parameter is assumed to be the type of the column in the target table.

```
UPDATE t2 SET c2 =? -- type is assumed to be type of c2
```

13. Dynamic parameters are allowed as the operand of the unary operators - or +. For example:

```
CREATE TABLE t1 (c11 INT, c12 SMALLINT, c13 DOUBLE, c14 CHAR(3))
SELECT * FROM t1 WHERE c11 BETWEEN -? AND +?
-- The type of both of the unary operators is INT
-- based on the context in which they are used (that is,
-- because c11 is INT, the unary parameters also get the
-- type INT.
```

14. LENGTH allow a dynamic parameter. The type is assumed to be a maximum length VARCHAR type.

```
SELECT LENGTH(?)
```

15. Qualified comparisons.

```
? = SOME (SELECT 1 FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type
1 = SOME (SELECT ? FROM t)
-- is valid. Dynamic parameter assumed to be INTEGER type.
```

16. A dynamic parameter is allowed as the left operand of an IS expression and is assumed to be a boolean.

Once the type of a dynamic parameter is determined based on the expression it is in, that expression is allowed anywhere it would normally be allowed if it did not include a dynamic parameter.

JOIN operations

The JOIN operations, which are among the possible *TableExpressions* in a FROM clause, perform joins between two tables. (You can also perform a join between two tables using an explicit equality test in a WHERE clause, such as "WHERE t1.col1 = t2.col2".)

Syntax

```
JOIN Operation
```

The JOIN operations are:

- INNER JOIN operation

  Specifies a join between two tables with an explicit join clause. See INNER JOIN operation.
- LEFT OUTER JOIN operation

  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table. See LEFT OUTER JOIN operation.
- RIGHT OUTER JOIN operation

  Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table. See RIGHT OUTER JOIN operation.

In all cases, you can specify additional restrictions on one or both of the tables being joined in outer join clauses or in the WHERE clause.

JOIN expressions and query optimization

For information on which types of joins are optimized, see *Tuning Java DB*.

INNER JOIN operation

An INNER JOIN is a JOIN operation that allows you to specify an explicit join clause.

Syntax

```
TableExpression [ INNER ] JOIN TableExpression { ON booleanExpression }
```

You can specify the join clause by specifying ON with a boolean expression.

The scope of expressions in the ON clause includes the current tables and any tables in outer query blocks to the current SELECT. In the following example, the ON clause refers to the current tables:

```
SELECT *
FROM SAMP.EMPLOYEE INNER JOIN SAMP.STAFF
ON EMPLOYEE.SALARY < STAFF.SALARY
```

The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

```
-- Join the EMP_ACT and EMPLOYEE tables
-- select all the columns from the EMP_ACT table and
-- add the employee's surname (LASTNAME) from the EMPLOYEE table
-- to each row of the result
SELECT SAMP.EMP_ACT.*, LASTNAME
     FROM SAMP.EMP_ACT JOIN SAMP.EMPLOYEE
     ON EMP_ACT.EMPNO = EMPLOYEE.EMPNO
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the
-- DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930.
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
     FROM SAMP.EMPLOYEE JOIN SAMP.DEPARTMENT
     ON WORKDEPT = DEPTNO
     AND YEAR(BIRTHDATE) < 1930

-- Another example of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X"
```

```
-- having 2 columns "R1" and "R2" and 1 row of data
SELECT *
FROM (VALUES (3, 4), (1, 5), (2, 6))
AS VALUESTABLE1(C1, C2)
JOIN (VALUES (3, 2), (1, 2),
(0, 3)) AS VALUESTABLE2(c1, c2)
ON VALUESTABLE1.c1 = VALUESTABLE2.c1
-- This results in:
-- C1          |C2          |C1          |2
-- -----------------------------------------------
-- 3           |4           |3           |2
-- 1           |5           |1           |2


-- List every department with the employee number and
-- last name of the manager

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
 FROM DEPARTMENT INNER JOIN EMPLOYEE
 ON MGRNO = EMPNO

-- List every employee number and last name
-- with the employee number and last name of their manager
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
 FROM EMPLOYEE E INNER JOIN
 DEPARTMENT INNER JOIN EMPLOYEE M
        ON MGRNO = M.EMPNO
        ON E.WORKDEPT = DEPTNO
```

LEFT OUTER JOIN operation

A LEFT OUTER JOIN is one of the JOIN operations that allow you to specify a join clause. It preserves the unmatched rows from the first (left) table, joining them with a NULL row in the shape of the second (right) table.

Syntax

```
TableExpression LEFT [ OUTER ] JOIN TableExpression
{
    ON booleanExpression
    }
```

The scope of expressions in either the ON clause includes the current tables and any tables in query blocks outer to the current SELECT. The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

Example 1

```
--match cities to countries in Asia

SELECT CITIES.COUNTRY, CITIES.CITY_NAME, REGION
FROM Countries
LEFT OUTER JOIN Cities
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE REGION = 'Asia'

-- use the synonymous syntax, LEFT JOIN, to achieve exactly
-- the same results as in the example above

SELECT  COUNTRIES.COUNTRY, CITIES.CITY_NAME,REGION
FROM COUNTRIES
LEFT JOIN CITIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE REGION = 'Asia'
```

Example 2

```
-- Join the EMPLOYEE and DEPARTMENT tables,
-- select the employee number (EMPNO),
-- employee surname (LASTNAME),
-- department number (WORKDEPT in the EMPLOYEE table
-- and DEPTNO in the DEPARTMENT table)
-- and department name (DEPTNAME)
-- of all employees who were born (BIRTHDATE) earlier than 1930

SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
   FROM SAMP.EMPLOYEE LEFT OUTER JOIN SAMP.DEPARTMENT
   ON WORKDEPT = DEPTNO
   AND YEAR(BIRTHDATE) < 1930

-- List every department with the employee number and
-- last name of the manager,
-- including departments without a manager

SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
      FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
     ON MGRNO = EMPNO
```

RIGHT OUTER JOIN operation

A RIGHT OUTER JOIN is one of the JOIN operations that allow you to specify a JOIN clause. It preserves the unmatched rows from the second (right) table, joining them with a NULL in the shape of the first (left) table. A LEFT OUTER JOIN B is equivalent to B RIGHT OUTER JOIN A, with the columns in a different order.

Syntax

```
TableExpression RIGHT [ OUTER ] JOIN TableExpression
{
    ON booleanExpression
    }
```

The scope of expressions in the ON clause includes the current tables and any tables in query blocks outer to the current SELECT. The ON clause can reference tables not being joined and does not have to reference either of the tables being joined (though typically it does).

Example 1

```
-- get all countries and corresponding cities, including
-- countries without any cities

SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE

-- get all countries in Africa and corresponding cities, including
-- countries without any cities

SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT OUTER JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
WHERE Countries.region = 'Africa'

-- use the synonymous syntax, RIGHT JOIN, to achieve exactly
-- the same results as in the example above

SELECT COUNTRIES.COUNTRY, CITIES.CITY_NAME
FROM CITIES
RIGHT JOIN COUNTRIES
ON CITIES.COUNTRY_ISO_CODE = COUNTRIES.COUNTRY_ISO_CODE
```

```
WHERE Countries.region = 'Africa'
```

Example 2

```
-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
-- List every employee number and last name
-- with the employee number and last name of their manager

SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
 FROM EMPLOYEE E RIGHT OUTER JOIN
 DEPARTMENT RIGHT OUTER JOIN EMPLOYEE M
        ON MGRNO = M.EMPNO
        ON E.WORKDEPT = DEPTNO
```

SQL queries

Query

A query creates a virtual table based on existing tables or constants built into tables.

Syntax

```
{
    ( Query )
 |
    Query INTERSECT [ ALL | DISTINCT ]
 Query
 |
    Query EXCEPT [ ALL | DISTINCT ] Query |
    Query UNION [ ALL | DISTINCT ] Query |
SelectExpression | VALUES Expression
}
```

You can arbitrarily put parentheses around queries, or use the parentheses to control the order of evaluation of the INTERSECT, EXCEPT, or UNION operations. These operations are evaluated from left to right when no parentheses are present, with the exception of INTERSECT operations, which would be evaluated before any UNION or EXCEPT operations.

Duplicates in UNION, INTERSECT, and EXCEPT ALL results

The ALL and DISTINCT keywords determine whether duplicates are eliminated from the result of the operation. If you specify the DISTINCT keyword, then the result will have no duplicate rows. If you specify the ALL keyword, then there may be duplicates in the result, depending on whether there were duplicates in the input. DISTINCT is the default, so if you don't specify ALL or DISTINCT, the duplicates will be eliminated. For example, UNION builds an intermediate *ResultSet* with all of the rows from both queries and eliminates the duplicate rows before returning the remaining rows. UNION ALL returns all rows from both queries as the result.

Depending on which operation is specified, if the number of copies of a row in the left table is L and the number of copies of that row in the right table is R, then the number of duplicates of that particular row that the output table contains (assuming the ALL keyword is specified) is:

- UNION: (L + R).
- EXCEPT: the maximum of (L – R) and 0 (zero).
- INTERSECT: the minimum of L and R.

Examples

```
-- a Select expression
SELECT *
```

```
FROM ORG

-- a subquery
SELECT *
FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS

-- a subquery
SELECT *
FROM (SELECT CLASS_CODE FROM CL_SCHED) AS CS (CLASS_CODE)

-- a UNION
-- returns all rows from columns DEPTNUMB and MANAGER
-- in table ORG
-- and (1,2) and (3,4)
-- DEPTNUMB and MANAGER are smallint columns
SELECT DEPTNUMB, MANAGER
FROM ORG
UNION ALL
VALUES (1,2), (3,4)

-- a values expression
VALUES (1,2,3)
-- List the employee numbers (EMPNO) of all employees in the EMPLOYEE
 table
-- whose department number (WORKDEPT) either begins with 'E' or
-- who are assigned to projects in the EMP_ACT table
-- whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'
SELECT EMPNO
     FROM EMPLOYEE
     WHERE WORKDEPT LIKE 'E%'
  UNION
  SELECT EMPNO
     FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
-- Make the same query as in the previous example
-- and "tag" the rows from the EMPLOYEE table with 'emp' and
-- the rows from the EMP_ACT table with 'emp_act'.
-- Unlike the result from the previous example,
-- this query may return the same EMPNO more than once,
-- identifying which table it came from by the associated "tag"
SELECT EMPNO, 'emp'
     FROM EMPLOYEE
     WHERE WORKDEPT LIKE 'E%'
  UNION
  SELECT EMPNO, 'emp_act' FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
-- Make the same query as in the previous example,
-- only use UNION ALL so that no duplicate rows are eliminated
SELECT EMPNO
      FROM EMPLOYEE
      WHERE WORKDEPT LIKE 'E%'
  UNION ALL
  SELECT EMPNO
     FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
-- Make the same query as in the previous example,
-- only include an additional two employees currently not in any table
 and
-- tag these rows as "new"
 SELECT EMPNO, 'emp'
     FROM EMPLOYEE
     WHERE WORKDEPT LIKE 'E%'
  UNION
  SELECT EMPNO, 'emp_act'
     FROM EMP_ACT
     WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
  UNION
     VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')
```

ScalarSubquery

You can place a *ScalarSubquery* anywhere an *Expression* is permitted. A *ScalarSubquery* turns a *SelectExpression* result into a scalar value because it returns only a single row and column value.

The query must evaluate to a single row with a single column.

Sometimes also called an expression subquery.

Syntax

```
(Query)
```

Examples

```
-- avg always returns a single value, so the subquery is
-- a ScalarSubquery
SELECT NAME, COMM
  FROM STAFF
  WHERE EXISTS
    (SELECT AVG(BONUS + 800)
       FROM EMPLOYEE
       WHERE COMM < 5000
       AND EMPLOYEE.LASTNAME = UPPER(STAFF.NAME)
)
-- Introduce a way of "generating" new data values,
-- using a query which selects from a VALUES clause (which is an
-- alternate form of a fullselect).
-- This query shows how a table can be derived called "X" having
-- 2 columns "R1" and "R2" and 1 row of data.
SELECT R1,R2
 FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
```

TableSubquery

A *TableSubquery* is a subquery that returns multiple rows.

Unlike a *ScalarSubquery*, a *TableSubquery* is allowed only:

- as a *TableExpression* in a FROM clause
- with EXISTS, IN, or quantified comparisons.

When used as a *TableExpression* in a FROM clause, it can return multiple columns. When used with EXISTS, it returns multiple columns only if you use * to return the multiple columns.

When used with IN or quantified comparisons, it must return a single column.

Syntax

```
(Query)
```

Example

```
-- a subquery used as a TableExpression in a FROM clause
SELECT VirtualFlightTable.flight_ID
FROM
    (SELECT flight_ID, orig_airport, dest_airport
    FROM Flights
    WHERE (orig_airport = 'SFO' OR dest_airport = 'SCL') )
AS VirtualFlightTable
-- a subquery (values expression) used as a TableExpression
-- in a FROM clause
SELECT mycol1
FROM
    (VALUES (1, 2), (3, 4))
AS mytable (mycol1, mycol2)
```

```
-- a subquery used with EXISTS
SELECT *
FROM Flights
WHERE EXISTS
    (SELECT * FROM Flights WHERE dest_airport = 'SFO'
    AND orig_airport = 'GRU')
-- a subquery used with IN
SELECT flight_id, segment_number
FROM Flights
WHERE flight_id IN
    (SELECT flight_ID
    FROM Flights WHERE orig_airport = 'SFO'
    OR dest_airport = 'SCL')
-- a subquery used with a quantified comparison
SELECT NAME, COMM
FROM STAFF
WHERE COMM >
(SELECT AVG(BONUS + 800)
     FROM EMPLOYEE
     WHERE COMM < 5000)
```

Built-in functions

A built-in function is an expression in which an SQL keyword or special operator executes some operation. Built-in functions use keywords or special built-in operators. Built-ins are SQL92Identifiers and are case-insensitive. Note that escaped functions like TIMESTAMPADD and TIMESTAMPDIFF are only accessible using the JDBC escape function syntax, and can be found in JDBC escape syntax.

Standard built-in functions

The standard built-in functions supported in Derby are as follows:

- ABS or ABSVAL function
- ACOS function
- ASIN function
- ATAN function
- ATAN2 function
- BIGINT function
- CAST function
- CEIL or CEILING function
- CHAR function
- Concatenation operator
- COS function
- NULLIF expressions
- CURRENT_DATE function
- CURRENT ISOLATION function
- CURRENT_TIME function
- CURRENT_TIMESTAMP function
- CURRENT_USER function
- DATE function
- DAY function
- DEGREES function
- DOUBLE function
- EXP function
- FLOOR function
- HOUR function
- IDENTITY_VAL_LOCAL function
- INTEGER function

- LENGTH function
- LN or LOG function
- LOG10 function
- LOCATE function
- LCASE or LOWER function
- LTRIM function
- MINUTE function
- MOD function
- MONTH function
- PI function
- RADIANS function
- RTRIM function
- SECOND function
- SESSION_USER function
- SIN function
- SMALLINT function
- SQRT function
- SUBSTR function
- TAN function
- TIME function
- TIMESTAMP function
- TRIM function
- UCASE or UPPER function
- USER function
- VARCHAR function
- YEAR function

Aggregates (set functions)

This section describes aggregates (also described as *set functions* in ANSI SQL-92 and as *column functions* in some database literature). They provide a means of evaluating an expression over a set of rows. Whereas the other built-in functions operate on a single expression, aggregates operate on a set of values and reduce them to a single scalar value. Built-in aggregates can calculate the minimum, maximum, sum, count, and average of an expression over a set of values as well as count rows.

The built-in aggregates can operate on the data types shown in Permitted Data Types for Built-in Aggregates.

Table 7. Permitted Data Types for Built-in Aggregates

Function Name	All Types	Numeric Built-in Data Types
COUNT	X	X
MIN		X
MAX		X
AVG		X
SUM		X

Aggregates are permitted only in the following:

- A *SelectItem* in a *SelectExpression.*
- A HAVING clause.
- An ORDER BY clause (using an alias name) if the aggregate appears in the result of the relevant query block. That is, an alias for an aggregate is permitted

in an ORDER BY clause if and only if the aggregate appears in a *SelectItem* in a *SelectExpression.*

All expressions in *SelectItems* in the *SelectExpression* must be either aggregates or grouped columns (see GROUP BY clause). (The same is true if there is a HAVING clause without a GROUP BY clause.) This is because the *ResultSet* of a *SelectExpression* must be either a scalar (single value) or a vector (multiple values), but not a mixture of both. (Aggregates evaluate to a scalar value, and the reference to a column can evaluate to a vector.) For example, the following query mixes scalar and vector values and thus is not valid:

```
-- not valid
SELECT MIN(flying_time), flight_id
FROM Flights
```

Aggregates are not allowed on outer references (correlations). This means that if a subquery contains an aggregate, that aggregate cannot evaluate an expression that includes a reference to a column in the outer query block. For example, the following query is not valid because SUM operates on a column from the outer query:

```
SELECT c1
FROM t1
GROUP BY c1
HAVING c2 >
    (SELECT t2.x
    FROM t2
    WHERE t2.y = SUM(t1.c3))
```

A cursor declared on a *ResultSet* that includes an aggregate in the outer query block is not updatable.

Derby supports the following aggregates:

- AVG function
- COUNT function
- MAX function
- MIN function
- SUM function

ABS or ABSVAL function

ABS or ABSVAL returns the absolute value of a numeric expression. The return type is the type of parameter. All built-in numeric types are supported (DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, BIGINT, NUMERIC, REAL, and SMALLINT).

Syntax

```
ABS(NumericExpression)
```

```
-- returns 3
VALUES ABS(-3)
```

ACOS function

The ACOS function returns the arc cosine of a specified number.

The specified number is the cosine, in radians, of the angle that you want. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

The returned value, in radians, is in the range of zero (0) to pi. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ACOS ( number )
```

ASIN function

The ASIN function returns the arc sine of a specified number.

The specified number is the sine, in radians, of the angle that you want. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero with the same sign as the specified number.
- If the absolute value of the specified number is greater than 1, an exception is returned that indicates that the value is out of range (SQL state 22003).

The returned value, in radians, is in the range -pi/2 to pi/2. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ASIN ( number )
```

ATAN function

The ATAN function returns the arc tangent of a specified number.

The specified number is the tangent, in radians, of the angle that you want. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero with the same sign as the specified number.

The returned value, in radians, is in the range -pi/2 to pi/2. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ATAN ( number )
```

ATAN2 function

The ATAN2 function returns the arctangent, in radians, of the quotient of the two arguments.

Upon successful completion, the function returns the arc tangent of *y*/*x* in the range -*pi* to *pi* radians, where *y* is the first argument and *x* is the second argument. The specified numbers must be DOUBLE PRECISION numbers.

- If either argument is NULL, the result of the function is NULL.
- If the first argument is zero and the second argument is positive, the result of the function is zero.
- If the first argument is zero and the second argument is negative, the result of the function is the double value closest to *pi*.
- If the first argument is positive and the second argument is zero, the result is the double value closest to *pi*/2.

- If the first argument is negative and the second argument is zero, the result is the double value closest to -*pi*/2.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
ATAN2 ( y, x )
```

AVG function

AVG is an aggregate function that evaluates the average of an expression over a set of rows (see Aggregates (set functions)). AVG is allowed only on expressions that evaluate to numeric data types.

Syntax

```
AVG ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. ALL is the default value if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1.0, 1.0, 1.0, 1.0, and 2.0, AVG(col) returns a smaller value than AVG(DISTINCT col).

Only one DISTINCT aggregate expression per *SelectExpression* is allowed. For example, the following query is not valid:

```
SELECT AVG (DISTINCT flying_time), SUM (DISTINCT miles)
FROM Flights
```

The expression can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to an SQL-92 numeric data type. You can therefore call methods that evaluate to SQL-92 data types. If an expression evaluates to NULL, the aggregate skips that value.

The resulting data type is the same as the expression on which it operates (it will never overflow). The following query, for example, returns the INTEGER *1*, which might not be what you would expect:

```
SELECT AVG(c1)
FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)
```

CAST the expression to another data type if you want more precision:

```
SELECT AVG(CAST (c1 AS DOUBLE PRECISION))
FROM (VALUES (1), (1), (1), (1), (2)) AS myTable (c1)
```

BIGINT function

The BIGINT function returns a 64-bit integer representation of a number or character string in the form of an integer constant.

Syntax

```
BIGINT (CharacterExpression | NumericExpression )
```

CharacterExpression

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument

is a CharacterExpression, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

NumericExpression

An expression that returns a value of any built-in numeric data type. If the argument is a NumericExpression, the result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application:

```
SELECT BIGINT (EMPNO) FROM EMPLOYEE
```

CASE expressions

Use the CASE expressions for conditional expressions in Derby.

CASE expression syntax

You can place a CASE expression anywhere an expression is allowed. It chooses an expression to evaluate based on a boolean test.

```
CASE
  WHEN booleanExpression THEN thenExpression
  [ WHEN booleanExpression THEN thenExpression ]...
  ELSE elseExpression
END
```

ThenExpression and *elseExpression* are both expressions that must be type-compatible. For built-in types, this means that the types must be the same or a built-in broadening conversion must exist between the types.

```
-- returns 3
VALUES CASE WHEN 1=1 THEN 3 ELSE 4 END
```

```
-- returns 7
VALUES
   CASE
      WHEN 1 = 2 THEN 3
      WHEN 4 = 5 THEN 6
      ELSE 7
   END
```

CAST function

The CAST function converts a value from one data type to another and provides a data type to a dynamic parameter (?) or a NULL value.

CAST expressions are permitted anywhere expressions are permitted.

Syntax

```
CAST ( [ Expression | NULL | ? ]
    AS Datatype)
```

The data type to which you are casting an expression is the *target type*. The data type of the expression from which you are casting is the *source type*.

CAST conversions among SQL-92 data types

The following table shows valid explicit conversions between source types and target types for SQL data types. This table shows which explicit conversions between data types are valid. The first column on the table lists the source data types. The first row lists the target data types. A "Y" indicates that a conversion from the source to the target is valid. For example, the first cell in the second row lists the source data type SMALLINT. The remaining cells on the second row indicate the whether or not you can convert SMALLINT to the target data types that are listed in the first row of the table.

Table 8. Explicit conversions between source types and target types for SQL data types

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP	XML
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
FLOAT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y	-
VARCHAR	Y	Y	Y	Y	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y	-
LONG VARCHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-
CHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-	-

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP	XML
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-	-
LONG VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	Y	-	-	-	-
CLOB	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-
DATE	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-
TIMESTAMP	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

If a conversion is valid, CASTs are allowed. Size incompatibilities between the source and target types might cause runtime errors.

Notes

In this discussion, the Derby SQL-92 data types are categorized as follows:

- *numeric*
  - Exact numeric (SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC)
  - Approximate numeric (FLOAT, REAL, DOUBLE PRECISION)
- *string*
  - Character string (CLOB, CHAR, VARCHAR, LONG VARCHAR)
  - Bit string (BLOB, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA)
- *date/time*

- DATE
- TIME
- TIMESTAMP

Conversions from numeric types

A numeric type can be converted to any other numeric type. If the target type cannot represent the non-fractional component without truncation, an exception is raised. If the target numeric cannot represent the fractional component (scale) of the source numeric, then the source is silently truncated to fit into the target. For example, casting 763.1234 as INTEGER yields 763.

Conversions from and to bit strings

Bit strings can be converted to other bit strings, but not character strings. Strings that are converted to bit strings are padded with trailing zeros to fit the size of the target bit string. The BLOB type is more limited and requires explicit casting. In most cases the BLOB type cannot be casted to and from other types.

Conversions of date/time values

A date/time value can always be converted to and from a TIMESTAMP. If a DATE is converted to a TIMESTAMP, the TIME component of the resulting TIMESTAMP is always 00:00:00. If a TIME data value is converted to a TIMESTAMP, the DATE component is set to the value of CURRENT_DATE at the time the CAST is executed. If a TIMESTAMP is converted to a DATE, the TIME component is silently truncated. If a TIMESTAMP is converted to a TIME, the DATE component is silently truncated.

```
SELECT CAST (miles AS INT)
FROM Flights
-- convert timestamps to text
INSERT INTO mytable (text_column)
VALUES (CAST (CURRENT_TIMESTAMP AS VARCHAR(100)))
-- you must cast NULL as a data type to use it
SELECT airline
FROM Airlines
UNION ALL
VALUES (CAST (NULL AS CHAR(2)))
-- cast a double as a decimal
SELECT CAST (FLYING_TIME AS DECIMAL(5,2))
FROM FLIGHTS
-- cast a SMALLINT to a BIGINT
VALUES CAST (CAST (12 as SMALLINT) as BIGINT)
```

Conversions of XML values

An XML value cannot be converted to any non-XML type using an explicit or implicit CAST. Use the XMLSERIALIZE operator to convert an XML type to a character type.

CEIL or CEILING function

The CEIL and CEILING functions round the specified number up, and return the smallest number that is greater than or equal to the specified number.

The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of these functions is NULL.
- If the specified number is equal to a mathematical integer, the result of these functions is the same as the specified number.
- If the specified number is zero (0), the result of these functions is zero.
- If the specified number is less than zero but greater than -1.0, then the result of these functions is zero.

The returned value is the smallest (closest to negative infinity) double floating point value that is greater than or equal to the specified number. The returned value is equal to a mathematical integer. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
CEIL ( number )
```

```
CEILING ( number )
```

CHAR function

The CHAR function returns a fixed-length character string representation.

The representations are:

- A character string, if the first argument is any type of character string.
- A datetime value, if the first argument is a date, time, or timestamp.
- A decimal number, if the first argument is a decimal number.
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL.
- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT.

The first argument must be of a built-in data type. The result of the CHAR function is a fixed-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value. The first argument cannot be an XML value. To convert an XML value to a CHAR of a specified length, you must use the SQL/XML serialization operator XMLSERIALIZE.

Character to character syntax

```
CHAR (CharacterExpression [, integer] )
```

CharacterExpression
: An expression that returns a value that is CHAR, VARCHAR, LONG VARCHAR, or CLOB data type.

integer
: The length attribute for the resulting fixed length character string. The value must be between 0 and 254.

If the length of the character-expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character-expression is greater than the length attribute of the result, truncation is performed. A warning is returned unless the truncated characters were all blanks and the character-expression was not a long string (LONG VARCHAR or CLOB).

Integer to character syntax

```
CHAR (IntegerExpression )
```

IntegerExpression
: An expression that returns a value that is an integer data type (either SMALLINT, INTEGER or BIGINT).

The result is the character string representation of the argument in the form of an SQL integer constant. The result consists of n characters that are the significant digits that represent the value of the argument with a preceding minus sign if the argument is negative. It is left justified.

- If the first argument is a small integer: The length of the result is 6. If the number of characters in the result is less than 6, then the result is padded on the right with blanks to length 6.

- If the first argument is a large integer: The length of the result is 11. If the number of characters in the result is less than 11, then the result is padded on the right with blanks to length 11.
- If the first argument is a big integer: The length of the result is 20. If the number of characters in the result is less than 20, then the result is padded on the right with blanks to length 20.

Datetime to character syntax

```
CHAR (DatetimeExpression )
```

DatetimeExpression

An expression that is one of the following three data types:

- **date**: The result is the character representation of the date. The length of the result is 10.
- **time**: The result is the character representation of the time. The length of the result is 8.
- **timestamp**: The result is the character string representation of the timestamp. The length of the result is 26.

Decimal to character

```
CHAR (DecimalExpression )
```

DecimalExpression

An expression that returns a value that is a decimal data type. If a different precision and scale is desired, the DECIMAL scalar function can be used first to make the change.

Floating point to character syntax

```
CHAR (FloatingPointExpression )
```

FloatingPointExpression

An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

Use the CHAR function to return the values for EDLEVEL (defined as smallint) as a fixed length character string:

```
SELECT CHAR(EDLEVEL) FROM EMPLOYEE
```

An EDLEVEL of 18 would be returned as the CHAR(6) value '18   ' (18 followed by four blanks).

Concatenation operator

The concatenation operator, **||**, concatenates its right operand to the end of its left operand. It operates on a character or bit expression.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

Syntax

```
{
    { CharacterExpression || CharacterExpression } |
    { BitExpression || BitExpression }
}
```

For character strings, if both the left and right operands are of type CHAR, the resulting type is CHAR; otherwise, it is VARCHAR. The normal blank padding/trimming rules for CHAR and VARCHAR apply to the result of this operator.

The length of the resulting string is the sum of the lengths of both operands.

For bit strings, if both the left and the right operands are of type CHAR FOR BIT DATA, the resulting type is CHAR FOR BIT DATA; otherwise, it is VARCHAR FOR BIT DATA.

```
--returns 'supercalifragilisticexbealidocious(sp?)'
VALUES 'supercalifragilistic' || 'exbealidocious' || '(sp?)'
-- returns NULL
VALUES CAST (null AS VARCHAR(7))|| 'AString'
-- returns '130asdf'
VALUES '130' || 'asdf'
```

COS function

The COS function returns the cosine of a specified number.

The specified number is the angle, in radians, that you want the cosine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.

Syntax

```
COS ( number )
```

COSH function

The COSH function returns the hyperbolic cosine of a specified number.

The specified number is the angle, in radians, that you want the hyperbolic cosine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is one (1.0).

Syntax

```
COSH ( number )
```

COT function

The COT function returns the cotangens of a specified number.

The specified number is the angle, in radians, that you want the cotangens for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.

Syntax

```
COT ( number )
```

COUNT function

COUNT is an aggregate function that counts the number of rows accessed in an expression (see Aggregates (set functions)). COUNT is allowed on all types of expressions.

Syntax

```
COUNT ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT qualifier eliminates duplicates. The ALL qualifier retains duplicates. ALL is assumed if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1, 1, 1, 1, and 2, COUNT(col) returns a greater value than COUNT(DISTINCT col).

Only one DISTINCT aggregate expression per *SelectExpression* is allowed. For example, the following query is not allowed:

```
-- query not allowed
SELECT COUNT (DISTINCT flying_time), SUM (DISTINCT miles)
FROM Flights
```

An *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. If an *Expression* evaluates to NULL, the aggregate is not processed for that value.

The resulting data type of COUNT is INTEGER.

```
-- Count the number of countries in each region,
-- show only regions that have at least 2
SELECT COUNT (country), region
FROM Countries
GROUP BY region
HAVING COUNT (country) > 1
```

COUNT(*) function

COUNT(*) is an aggregate function that counts the number of rows accessed. No NULLs or duplicates are eliminated. COUNT(*) does not operate on an expression.

Syntax

```
COUNT(*)
```

The resulting data type is INTEGER.

```
-- Count the number of rows in the Flights table
SELECT COUNT(*)
FROM Flights
```

CURRENT DATE function

CURRENT DATE is a synonym for CURRENT_DATE.

CURRENT_DATE function

CURRENT_DATE returns the current date; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax

```
CURRENT_DATE
```

or, alternately

```
CURRENT DATE
```

```
-- find available future flights:
SELECT * FROM Flightavailability where flight_date > CURRENT_DATE;
```

CURRENT ISOLATION function

CURRENT ISOLATION returns the current isolation level as a char(2) value of either ""(blank), "UR", "CS", "RS", or "RR".

Syntax

```
CURRENT ISOLATION
```

```
VALUES CURRENT ISOLATION
```

CURRENT_ROLE function

CURRENT_ROLE returns the authorization identifier of the current role. If there is no current role, it returns NULL.

This function returns a string of up to 258 characters. This is twice the length of an identifier (128*2) + 2, to allow for quoting.

Syntax

```
CURRENT_ROLE
```

Example

```
VALUES CURRENT_ROLE
```

CURRENT SCHEMA function

CURRENT SCHEMA returns the schema name used to qualify unqualified database object references.

Note: CURRENT SCHEMA and CURRENT SQLID are synonyms.

These functions return a string of up to 128 characters.

Syntax

```
CURRENT SCHEMA

-- or, alternately:

CURRENT SQLID
```

```
-- Set the name column default to the current schema:
CREATE TABLE mytable (id int, name VARCHAR(128) DEFAULT CURRENT SQLID)
-- Inserts default value of current schema value into the table:
INSERT INTO mytable(id) VALUES (1)
-- Returns the rows with the same name as the current schema:
SELECT name FROM mytable WHERE name = CURRENT SCHEMA
```

CURRENT TIME function

CURRENT TIME is a synonym for CURRENT_TIME.

CURRENT_TIME function

CURRENT_TIME returns the current time; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax

```
CURRENT_TIME
```

or, alternately

```
CURRENT TIME
```

```
VALUES CURRENT_TIME
-- or, alternately:

VALUES CURRENT TIME
```

CURRENT TIMESTAMP function

CURRENT TIMESTAMP is a synonym for CURRENT_TIMESTAMP.

CURRENT_TIMESTAMP function

CURRENT_TIMESTAMP returns the current timestamp; the value returned does not change if it is executed more than once in a single statement. This means the value is fixed even if there is a long delay between fetching rows in a cursor.

Syntax

```
CURRENT_TIMESTAMP
```

or, alternately

```
CURRENT TIMESTAMP
```

```
VALUES CURRENT_TIMESTAMP
-- or, alternately:

VALUES CURRENT TIMESTAMP
```

CURRENT_USER function

CURRENT_USER returns the authorization identifier of the current user (the name of the user passed in when the user connected to the database). If there is no current user, it returns *APP.*

USER and SESSION_USER are synonyms.

These functions return a string of up to 128 characters.

Syntax

```
CURRENT_USER
```

```
VALUES CURRENT_USER
```

DATE function

The DATE function returns a date from a value.

The argument must be a date, timestamp, a positive number less than or equal to 2,932,897, a valid string representation of a date or timestamp, or a string of length 7 that is not a CLOB, LONG VARCHAR, or XML value. If the argument is a string of length 7, it must represent a valid date in the form yyyynnn, where yyyy are digits denoting a year, and nnn are digits between 001 and 366, denoting a day of that year. The result of the

function is a date. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp: The result is the date part of the value.
- If the argument is a number: The result is the date that is n-1 days after January 1, 0001, where n is the integral part of the number.
- If the argument is a string with a length of 7: The result is the date represented by the string.

Syntax

```
DATE ( expression )
```

This example results in an internal representation of '1988-12-25'.

```
VALUES DATE('1988-12-25')
```

DAY function

The DAY function returns the day part of a value.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 1 and 31. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
DAY ( expression )
```

Example

```
values day('2007-08-02');
```

The resulting value is 2.

DEGREES function

The DEGREES function converts a specified number from radians to degrees.

The specified number is an angle measured in radians, which is converted to an approximately equivalent angle measured in degrees. The specified number must be a DOUBLE PRECISION number.

Attention: The conversion from radians to degrees is not exact. You should not expect DEGREES(ACOS(0.5)) to return exactly 60.0.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
DEGREES ( number )
```

DOUBLE function

The DOUBLE function returns a floating-point number corresponding to a:

- number if the argument is a numeric expression.
- character string representation of a number if the argument is a string expression.

Numeric to double

```
DOUBLE [PRECISION] (NumericExpression )
```

NumericExpression
The argument is an expression that returns a value of any built-in numeric data type.

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value. The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable.

Character string to double

```
DOUBLE (StringExpression )
```

StringExpression
The argument can be of type CHAR or VARCHAR in the form of a numeric constant. Leading and trailing blanks in argument are ignored.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value. The result is the same number that would occur if the string was considered a constant and assigned to a double-precision floating-point column or variable.

EXP function

The EXP function returns e raised to the power of the specified number.

The specified number is the exponent that you want to raise e to. The specified number must be a DOUBLE PRECISION number.

The constant e is the base of the natural logarithms.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
EXP ( number )
```

FLOOR function

The FLOOR function rounds the specified number down, and returns the largest number that is less than or equal to the specified number.

The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is equal to a mathematical integer, the result of this function is the same as the specified number.
- If the specified number is zero (0), the result of this function is zero.

The returned value is the largest (closest to positive infinity) double floating point value that is less than or equal to the specified number. The returned value is equal to a mathematical integer. The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
FLOOR ( number )
```

HOUR function

The HOUR function returns the hour part of a value.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 0 and 24. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
HOUR ( expression )
```

Example

Select all the classes that start in the afternoon from a table called TABLE1.

```
SELECT * FROM TABLE1 WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

IDENTITY_VAL_LOCAL function

Derby supports the IDENTITY_VAL_LOCAL function.

Syntax:

```
 IDENTITY_VAL_LOCAL ( )
```

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value of an identity column for a connection, where the assignment occurred as a result of a single row INSERT statement using a VALUES clause.

The IDENTITY_VAL_LOCAL function has no input parameters. The result is a DECIMAL (31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the IDENTITY_VAL_LOCAL function, for a connection, is the value assigned to the identity column of the table identified in the most recent single row INSERT statement. The INSERT statement must contain a VALUES clause on a table containing an identity column. The assigned value is an identity value generated by Derby. The function returns a null value when a single row INSERT statement with a VALUES clause has not been issued for a table containing an identity column.

The result of the function is not affected by the following:

- A single row INSERT statement with a VALUES clause for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- An INSERT statement with a fullselect

If a table with an identity column has an INSERT trigger defined that inserts into another table with another identity column, then the IDENTITY_VAL_LOCAL() function will return the generated value for the statement table, and not for the table modified by the trigger.

Examples:

```
ij> create table t1(c1 int generated always as identity, c2 int);
0 rows inserted/updated/deleted
ij> insert into t1(c2) values (8);
1 row inserted/updated/deleted
ij> values IDENTITY_VAL_LOCAL();
1
-------------------------------
1
1 row selected
ij> select IDENTITY_VAL_LOCAL()+1, IDENTITY_VAL_LOCAL()-1 from t1;
1                                |2
-------------------------------------------------------------------
2                                |0
1 row selected
ij> insert into t1(c2) values (IDENTITY_VAL_LOCAL());
```

```
1 row inserted/updated/deleted
ij> select * from t1;
C1             |C2
-------------------------------
1              |8
2              |1
2 rows selected
ij> values IDENTITY_VAL_LOCAL();
1
-------------------------------
2
1 row selected
ij> insert into t1(c2) values (8), (9);
2 rows inserted/updated/deleted
ij> -- multi-values insert, return value of the function should not
 change
values IDENTITY_VAL_LOCAL();
1
-------------------------------
2
1 row selected
ij> select * from t1;
C1             |C2
-------------------------------
1              |8
2              |1
3              |8
4              |9
4 rows selected
ij> insert into t1(c2) select c1 from t1;
4 rows inserted/updated/deleted
-- insert with sub-select, return value should not change
ij> values IDENTITY_VAL_LOCAL();
1
-------------------------------
2
1 row selected
ij> select * from t1;
C1             |C2
-------------------------------
1              |8
2              |1
3              |8
4              |9
5              |1
6              |2
7              |3
8              |4
8 rows selected
```

INTEGER function

The INTEGER function returns an integer representation of a number, character string, date, or time in the form of an integer constant.

Syntax

```
INT[EGER] (NumericExpression | CharacterExpression )
```

NumericExpression
An expression that returns a value of any built-in numeric data type. If the argument is a numeric-expression, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

CharacterExpression

An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. The character string cannot be a long string. If the argument is a character-expression, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value:

```
SELECT INTEGER (SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
FROM EMPLOYEE
ORDER BY 1 DESC
```

LCASE or LOWER function

LCASE or LOWER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to lowercase.

Syntax

```
LCASE or LOWER ( CharacterExpression )
```

A *CharacterExpression* is a CHAR, VARCHAR, or LONG VARCHAR data type or any built-in type that is implicitly converted to a string (except a bit expression).

If the parameter type is CHAR or LONG VARCHAR, the return type is CHAR or LONG VARCHAR. Otherwise, the return type is VARCHAR.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

If the *CharacterExpression* evaluates to null, this function returns null.

```
-- returns 'asd1#w'
VALUES LOWER('aSD1#w')

SELECT LOWER(flight_id) FROM Flights
```

LENGTH function

LENGTH is applied to either a character string expression or a bit string expression and returns the number of characters in the result.

Because all built-in data types are implicitly converted to strings, this function can act on all built-in data types.

Syntax

```
LENGTH ( { CharacterExpression | BitExpression } )
```

```
-- returns 20
VALUES LENGTH('supercalifragilistic')
-- returns 1
VALUES LENGTH(X'FF')
-- returns 4
VALUES LENGTH(1234567890)
```

LN or LOG function

The LN and LOG functions return the natural logarithm (base e) of the specified number.

The specified number must be a DOUBLE PRECISION number that is greater than zero (0).

- If the specified number is NULL, the result of these functions is NULL.
- If the specified number is zero or a negative number, an exception is returned that indicates that the value is out of range (SQL state 22003).

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
LN ( number )
```

```
LOG ( number )
```

LOG10 function

The LOG10 function returns the base-10 logarithm of the specified number.

The specified number must be a DOUBLE PRECISION number that is greater than zero (0).

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero or a negative number, an exception is returned that indicates that the value is out of range (SQL state 22003).

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
LOG10 ( number )
```

LOCATE function

The LOCATE function is used to search for a string within another string. If the desired string is found, LOCATE returns the index at which it is found. If the desired string is not found, LOCATE returns 0.

Syntax

```
LOCATE(CharacterExpression, CharacterExpression [, StartPosition] )
```

There are two required arguments to the LOCATE function, and a third optional argument.

- The first CharacterExpression specifies the string to search **for**.
- The second CharacterExpression specifies the string **in which** to search.
- The third argument is the startPosition, and specifies the position in the second argument at which the search is to start. If the third argument is not provided, the LOCATE function starts its search at the beginning of the second argument.

The return type for LOCATE is an integer. The LOCATE function returns an integer indicating the index position within the second argument at which the first argument was first located. Index positions start with 1. If the first argument is not found in the second argument, LOCATE returns 0. If the first argument is an empty string (''), LOCATE returns the value of the third argument (or 1 if it was not provided), even if the second argument is also an empty string. If a NULL value is passed for either of the CharacterExpression arguments, NULL is returned.

```
-- returns 2, since 'love' is found at index position 2:
            VALUES LOCATE('love', 'clover')
```

```
-- returns 0, since 'stove' is not found in 'clover':
            VALUES LOCATE('stove', 'clover')
```

```
-- returns 5 (note the start position is 4):
            VALUES LOCATE('iss', 'Mississippi', 4)
```

```
-- returns 1, because the empty string is a special case:
            VALUES LOCATE('', 'ABC')
```

```
-- returns 0, because 'AAA' is not found in '':
            VALUES LOCATE('AAA', '')
```

```
-- returns 3
            VALUES LOCATE('', '', 3)
```

LTRIM function

LTRIM removes blanks from the beginning of a character string expression.

Syntax

```
LTRIM(CharacterExpression)
```

A *CharacterExpression* is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string.

LTRIM returns NULL if *CharacterExpression* evaluates to null.

```
-- returns 'asdf  '
VALUES LTRIM(' asdf  ')
```

MAX function

MAX is an aggregate function that evaluates the maximum of an expression over a set of rows (see Aggregates (set functions)). MAX is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, CHAR FOR BIT DATA, etc.).

Syntax

```
MAX ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT and ALL qualifiers eliminate or retain duplicates, but these qualifiers have no effect in a MAX expression. Only one DISTINCT aggregate expression per *SelectExpression* is allowed. For example, the following query is not allowed:

```
SELECT COUNT (DISTINCT flying_time), MAX (DISTINCT miles)
FROM Flights
```

The *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a *java.lang.Integer* or *int* evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

The type's comparison rules determine the maximum value. For CHAR and VARCHAR, the number of blank spaces at the end of the value can affect how MAX is evaluated. For example, if the values 'z' and 'z ' are both stored in a column, you cannot control which one will be returned as the maximum, because blank spaces are ignored for character comparisons.

The resulting data type is the same as the expression on which it operates (it will never overflow).

```
-- find the latest date in the FlightAvailability table
SELECT MAX (flight_date) FROM FlightAvailability
-- find the longest flight originating from each airport,
-- but only when the longest flight is over 10 hours
SELECT MAX(flying_time), orig_airport
FROM Flights
GROUP BY orig_airport
HAVING MAX(flying_time) > 10
```

MIN function

MIN is an aggregate function that evaluates the minimum of an expression over a set of rows (see Aggregates (set functions)). MIN is allowed only on expressions that evaluate to built-in data types (including CHAR, VARCHAR, DATE, TIME, etc.).

Syntax

```
MIN ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT and ALL qualifiers eliminate or retain duplicates, but these qualifiers have no effect in a MIN expression. Only one DISTINCT aggregate expression per *SelectExpression* is allowed. For example, the following query is not allowed:

```
SELECT COUNT (DISTINCT flying_time), MIN (DISTINCT miles)
FROM Flights
```

The *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in data type. You can therefore call methods that evaluate to built-in data types. (For example, a method that returns a *java.lang.Integer* or *int* evaluates to an INTEGER.) If an expression evaluates to NULL, the aggregate skips that value.

The type's comparison rules determine the minimum value. For CHAR and VARCHAR, the number of blank spaces at the end of the value can affect how MIN is evaluated. For example, if the values 'z' and 'z ' are both stored in a column, you cannot control which one will be returned as the minimum, because blank spaces are ignored for character comparisons.

The resulting data type is the same as the expression on which it operates (it will never overflow).

```
-- NOT valid:
SELECT DISTINCT flying_time, MIN(DISTINCT miles) from Flights
-- valid:
SELECT COUNT(DISTINCT flying_time), MIN(DISTINCT miles) from Flights
-- find the earliest date:
SELECT MIN (flight_date) FROM FlightAvailability;
```

MINUTE function

The MINUTE function returns the minute part of a value.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 0 and 59. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
MINUTE ( expression )
```

Example

Select all rows from the "flights" table where the "departure_time" is between 6:00 and 6:30 AM:

```
SELECT * FROM flights WHERE HOUR(departure_time) = 6 and
 MINUTE(departure_time) < 31;
```

MOD function

MOD returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative.

Syntax

```
mod(integer_type, integer_type)
```

The result of the function is:
- SMALLINT if both arguments are SMALLINT.
- INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT.
- BIGINT if one integer is BIGINT and the other argument is BIGINT, INTEGER, or SMALLINT.

The result can be null; if any argument is null, the result is the null value.

MONTH function

The MONTH function returns the month part of a value.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 1 and 12. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
MONTH ( expression )
```

Example

Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

```
SELECT * FROM EMPLOYEE WHERE MONTH(BIRTHDATE) = 12
```

NULLIF expressions

Use the NULLIF expressions for conditional expressions in Derby.

NULLIF expression syntax

```
NULLIF ( L, R )
```

The NULLIF expression is very similar to the CASE expression. For example:

```
NULLIF(V1,V2)
```

is equivalent to the following CASE expression:

```
CASE WHEN V1=V2 THEN NULL ELSE V1 END
```

PI function

The PI function returns a value that is closer than any other value to pi.

The constant pi is the ratio of the circumference of a circle to the diameter of a circle.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
PI (  )
```

RADIANS function

The RADIANS function converts a specified number from degrees to radians.

The specified number is an angle measured in degrees, which is converted to an approximately equivalent angle measured in radians. The specified number must be a DOUBLE PRECISION number.

Attention: The conversion from degrees to radians is not exact.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
RADIANS ( number )
```

RANDOM function

The RANDOM function returns a random number.

The RANDOM function returns a DOUBLE PRECISION number with positive sign, greater than or equal to zero (0), and less than one (1.0).

Syntax

```
RANDOM()
```

RAND function

The RAND function returns a random number given a seed number

The RAND function returns a DOUBLE PRECISION number with positive sign, greater than or equal to zero (0), and less than one (1.0), given an INTEGER seed number.

Syntax

```
RAND( seed )
```

ROW_NUMBER function

The ROW_NUMBER function returns the row number over a named or unnamed window specification.

The ROW_NUMBER function does not take any arguments, and for each row over the window it returns an ever increasing BIGINT. It is normally used to limit the number of rows returned for a query. The LIMIT keyword used in other databases is not defined in the SQL standard, and is not supported.

- Derby does not currently allow the named or unnamed window specification to be specified in the OVER() clause, but requires an empty parenthesis. This means the function is evaluated over the entire result set.
- The ROW_NUMBER function cannot currently be used in a WHERE clause.
- Derby does not currently support ORDER BY in subqueries, so there is currently no way to guarantee the order of rows in the SELECT subquery. An optimizer override can be used to force the optimizer to use an index ordered on the desired column(s) if ordering is a firm requirement.

The data type of the returned value is a BIGINT number.

Syntax

```
ROW_NUMBER() OVER ()
```

Example

To limit the number of rows returned from a query to the 10 first rows of table T, use the following query:

```
SELECT * FROM (
   SELECT
     ROW_NUMBER() OVER () AS R,
     T.*
   FROM T
) AS TR
   WHERE R <= 10;
```

RTRIM function

RTRIM removes blanks from the end of a character string expression.

Syntax

```
RTRIM(CharacterExpression)
```

A CharacterExpression is a CHAR, VARCHAR, or LONG VARCHAR data type, any built-in type that is implicitly converted to a string.

RTRIM returns NULL if *CharacterExpression* evaluates to null.

```
-- returns ' asdf'
VALUES RTRIM(' asdf  ')
-- returns 'asdf'
VALUES RTRIM('asdf  ')
```

SECOND function

The SECOND function returns the seconds part of a value.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is not a CLOB, LONG VARCHAR, or XML value. The result of the function is an integer between 0 and 59. If the argument can be null, the result can be null. If the argument is null, the result is 0.

Syntax

```
SECOND ( expression )
```

Example

The RECEIVED column contains a timestamp that has an internal value equivalent to 2005-12-25-17.12.30.000000. To return only the seconds part of the timestamp, use the following syntax:

```
SECOND(RECEIVED)
```

The value 30 is returned.

SESSION_USER function

SESSION_USER returns the authorization identifier or name of the current user. If there is no current user, it returns *APP.*

USER, CURRENT_USER, and SESSION_USER are synonyms.

Syntax

```
SESSION_USER
```

```
VALUES SESSION_USER
```

SIGN function

The SIGN function returns the sign of the specified number.

The specified number is the number you want the sign of. The specified number must be a DOUBLE PRECISION number.

The data type of the returned value is INTEGER.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero (0).
- If the specified number is greater than zero (0), the result of this function is plus one (+1).
- If the specified number is less than zero (0), the result of this function is minus one (-1).

Syntax

```
SIGN ( number )
```

SIN function

The SIN function returns the sine of a specified number.

The specified number is the angle, in radians, that you want the sine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
SIN ( number )
```

SINH function

The SINH function returns the hyperbolic sine of a specified number.

The specified number is the angle, in radians, that you want the hyperbolic sine for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
SIN ( number )
```

SMALLINT function

The SMALLINT function returns a small integer representation of a number or character string in the form of a small integer constant.

Syntax

```
SMALLINT ( NumericExpression | CharacterExpression )
```

NumericExpression
An expression that returns a value of any built-in numeric data type. If the argument is a NumericExpression, the result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The decimal part of the argument is truncated if present.

CharacterExpression
An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant. However, the value of the constant must be in the range of small integers. The character string cannot be a long string. If the argument is a CharacterExpression, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

The result of the function is a small integer. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example
To determine the small integer representation of the number 32767.99, use this clause:

```
VALUES SMALLINT (32767.99)
```

The result is 32767.

To determine the small integer representation of the number 1, use this clause:

```
VALUES SMALLINT (1)
```

The result is 1.

SQRT function

Returns the square root of a floating point number; only the built-in types REAL, FLOAT, and DOUBLE PRECISION are supported. The return type for SQRT is the type of the parameter.

Note: To execute SQRT on other data types, you must cast them to floating point types.

Syntax

```
SQRT(FloatingPointExpression)
```

```
-- throws an exception if any row stores a negative number:
VALUES SQRT(3421E+09)

-- returns the square root of an INTEGER after casting it as a
-- floating point data type:
SELECT SQRT(myDoubleColumn) FROM MyTable

VALUES SQRT (CAST(25 AS FLOAT));
```

SUBSTR function

The SUBSTR function acts on a character string expression or a bit string expression. The type of the result is a VARCHAR in the first case and VARCHAR FOR BIT DATA in the second case. The length of the result is the maximum length of the source type.

Syntax

```
SUBSTR({ CharacterExpression },
   StartPosition [, LengthOfString ] )
```

The parameter *startPosition* and the optional parameter *lengthOfString* are both integer expressions. The first character or bit has a *startPosition* of 1. If you specify 0, Derby assumes that you mean 1.

The parameter *characterExpression* is a CHAR, VARCHAR, or LONG VARCHAR data type or any built-in type that is implicitly converted to a string (except a bit expression).

For character expressions, the *startPosition* and *lengthOfString* parameters refer to characters. For bit expressions, the *startPosition* and *lengthOfString* parameters refer to bits.

If the *startPosition* is positive, it refers to position from the start of the source expression (counting the first character as 1). The *startPosition* cannot be a negative number.

If the *lengthOfString* is not specified, SUBSTR returns the substring of the expression from the *startPosition* to the end of the source expression. If *lengthOfString* is specified, SUBSTR returns a VARCHAR or VARBIT of length *lengthOfString* starting at the *startPosition*. The SUBSTR function returns an error if you specify a negative number for the parameter *lengthOfString*.

Examples

To return a substring of the word `hello`, starting at the second character and continuing until the end of the word, use the following clause:

```
VALUES SUBSTR('hello', 2)
```

The result is '`ello`'.

To return a substring of the word `hello`, starting at the first character and continuing for two characters, use the following clause:

```
VALUES SUBSTR('hello',1,2)
```

The result is 'he'.

SUM function

SUM is an aggregate function that evaluates the sum of the expression over a set of rows (see Aggregates (set functions)). SUM is allowed only on expressions that evaluate to numeric data types.

Syntax

```
SUM ( [ DISTINCT | ALL ] Expression )
```

The DISTINCT and ALL qualifiers eliminate or retain duplicates. ALL is assumed if neither ALL nor DISTINCT is specified. For example, if a column contains the values 1, 1, 1, 1, and 2, SUM(col) returns a greater value than SUM(DISTINCT col).

Only one DISTINCT aggregate expression per *SelectExpression* is allowed. For example, the following query is not allowed:

```
SELECT AVG (DISTINCT flying_time), SUM (DISTINCT miles)
FROM Flights
```

The *Expression* can contain multiple column references or expressions, but it cannot contain another aggregate or subquery. It must evaluate to a built-in numeric data type. If an expression evaluates to NULL, the aggregate skips that value.

The resulting data type is the same as the expression on which it operates (it might overflow).

```
-- find all economy seats available:
SELECT SUM (economy_seats) FROM Airlines;

-- use SUM on multiple column references
-- (find the total number of all seats purchased):
SELECT SUM (economy_seats_taken + business_seats_taken +
 firstclass_seats_taken)
as seats_taken FROM FLIGHTAVAILABILITY;
```

TAN function

The TAN function returns the tangent of a specified number.

The specified number is the angle, in radians, that you want the tangent for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
TAN ( number )
```

TANH function

The TANH function returns the hyperbolic tangent of a specified number.

The specified number is the angle, in radians, that you want the hyperbolic tangent for. The specified number must be a DOUBLE PRECISION number.

- If the specified number is NULL, the result of this function is NULL.
- If the specified number is zero (0), the result of this function is zero.

The data type of the returned value is a DOUBLE PRECISION number.

Syntax

```
TANH ( number )
```

TIME function

The TIME function returns a time from a value.

The argument must be a time, timestamp, or a valid string representation of a time or timestamp that is not a CLOB, LONG VARCHAR, or XML value. The result of the function is a time. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a time: The result is that time.
- If the argument is a timestamp: The result is the time part of the timestamp.
- If the argument is a string: The result is the time represented by the string.

Syntax

```
TIME ( expression )
```

```
values time(current_timestamp)
```

If the current time is 5:03 PM, the value returned is `17:03:00`.

TIMESTAMP function

The TIMESTAMP function returns a timestamp from a value or a pair of values.

The rules for the arguments depend on whether the second argument is specified:

- If only one argument is specified: It must be a timestamp, a valid string representation of a timestamp, or a string of length 14 that is not a CLOB, LONG VARCHAR, or XML value. A string of length 14 must be a string of digits that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.
- If both arguments are specified: The first argument must be a date or a valid string representation of a date and the second argument must be a time or a valid string representation of a time.

The other rules depend on whether the second argument is specified:

- If both arguments are specified: The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.
- If only one argument is specified and it is a timestamp: The result is that timestamp.
- If only one argument is specified and it is a string: The result is the timestamp represented by that string. If the argument is a string of length 14, the timestamp has a microsecond part of zero.

Syntax

```
TIMESTAMP ( expression [, expression ] )
```

Examples

The second column in table `records_table` contains dates (such as 1998-12-25) and the third column contains times of day (such as 17:12:30). You can return the timestamp with this statement:

```
SELECT TIMESTAMP(col2, col3) FROM records_table
```

The following clause returns the value `1998-12-25-17:12:30.0`:

```
VALUES TIMESTAMP('1998-12-25', '17.12.30');
1
--------------------------
1998-12-25 17:12:30.0
```

TRIM function

TRIM is a function that takes a character expression and returns that expression with leading and/or trailing pad characters removed. Optional parameters indicate whether leading, or trailing, or both leading and trailing pad characters should be removed, and specify the pad character that is to be removed.

Syntax

```
TRIM( [ trimOperands ] trimSource)
```

```
trimOperands  ::= { trimType [ trimCharacter ]  FROM | trimCharacter FROM
 }
trimType      ::= { LEADING | TRAILING | BOTH }
trimCharacter ::= CharacterExpression
trimSource    ::= CharacterExpression
```

If *trimType* is not specified, it will default to BOTH. If *trimCharacter* is not specified, it will default to the space character (' '). Otherwise the *trimCharacter* expression must evaulate to one of the following:

- a character string whose length is exactly one, or.
- NULL

If either *trimCharacter* or *trimSource* evaluates to NULL, the result of the TRIM function is NULL. Otherwise, the result of the TRIM function is defined as follows:

- If trimType is LEADING, the result will be the trimSource value with all leading occurrences of trimChar removed.
- If trimType is TRAILING, the result will be the trimSource value with all trailing occurrences of trimChar removed.
- If trimType is BOTH, the result will be the trimSource value with all leading *and* trailing occurrences of trimChar removed.

If *trimSource*'s data type is CHAR or VARCHAR, the return type of the TRIM function will be VARCHAR. Otherwise the return type of the TRIM function will be CLOB.

Examples

```
-- returns 'derby' (no spaces)
VALUES TRIM('  derby ')
```

```
-- returns 'derby' (no spaces)
VALUES TRIM(BOTH ' ' FROM '  derby ')
```

```
-- returns 'derby ' (with a space at the end)
VALUES TRIM(LEADING ' ' FROM '  derby ')
```

```
-- returns '  derby' (with two spaces at the beginning)
VALUES TRIM(TRAILING ' ' FROM '  derby ')
```

```
-- returns NULL
VALUES TRIM(cast (null as char(1)) FROM '  derby ')
```

```
-- returns NULL
VALUES TRIM(' ' FROM cast(null as varchar(30)))
```

```
-- returns ' derb' (with a space at the beginning)
VALUES TRIM('y' FROM ' derby')
```

```
-- results in an error because trimCharacter can only be 1 character
VALUES TRIM('by' FROM ' derby')
```

UCASE or UPPER function

UCASE or UPPER takes a character expression as a parameter and returns a string in which all alpha characters have been converted to uppercase.

Syntax

```
UCASE or UPPER ( CharacterExpression )
```

If the parameter type is CHAR , the return type is CHAR. Otherwise, the return type is VARCHAR.

Note: UPPER and LOWER follow the database locale. See *territory=ll_CC attribute* for more information about specifying locale.

The length and maximum length of the returned value are the same as the length and maximum length of the parameter.

Example

To return the string `aSD1#w` in uppercase, use the following clause:

```
VALUES UPPER('aSD1#w')
```

The value returned is `ASD1#W`.

USER function

USER returns the authorization identifier or name of the current user. If there is no current user, it returns *APP.*

USER, CURRENT_USER, and SESSION_USER are synonyms.

Syntax

```
USER
```

```
VALUES USER
```

VARCHAR function

The VARCHAR function returns a varying-length character string representation of a character string.

Character to varchar syntax

```
VARCHAR (CharacterStringExpression )
```

CharacterStringExpression

An expression whose value must be of a character-string data type with a maximum length of 32,672 bytes.

Datetime to varchar syntax

```
VARCHAR (DatetimeExpression )
```

DatetimeExpression

An expression whose value must be of a date, time, or timestamp data type.

Using the EMPLOYEE table, select the job description (JOB defined as CHAR(8)) for Dolores Quintana as a VARCHAR equivelant:

```
SELECT VARCHAR(JOB)
FROM EMPLOYEE
WHERE LASTNAME = 'QUINTANA'
```

XMLEXISTS operator

XMLEXISTS is an SQL/XML operator that you can use to query XML values in SQL.

The XMLEXISTS operator has two arguments, an XML query expression and a Derby XML value.

Syntax

```
XMLEXISTS ( xquery-string-literal
    PASSING BY REF xml-value-expression [ BY REF ] )
```

`xquery-string-literal`
Must be specified as a string literal. If this argument is specified as a parameter, an expression that is not a literal, or a literal that is not a string (for example an integer), Derby throws an error. The `xquery-string-literal` argument must also be an XPath expression that is supported by Apache Xalan. Derby uses Apache Xalan to evaluate all XML query expressions. Because Xalan does not support full XQuery, neither does Derby. If Xalan is unable to compile or execute the query argument, Derby catches the error that is thrown by Xalan and throws the error as a SQLException. For more on XPath and XQuery expressions, see these Web sites: http://www.w3.org/TR/xpath and http://www.w3.org/TR/xquery/.

`xml-value-expression`
Must be an XML data value and must constitute a well-formed SQL/XML document. The `xml-value-expression` argument cannot be a parameter. Derby does not perform implicit parsing nor casting of XML values, so use of strings or any other data type results in an error. If the argument is a sequence that is returned by the DerbyXMLQUERY operator, the argument is accepted if it is a sequence of exactly one node that is a document node. Otherwise Derby throws an error.

BY REF
Optional keywords that describe the only value passing mechanism supported by Derby. Since BY REF is also the default passing mechanism, the XMLEXISTS operator behaves the same whether the keywords are present or not. For more information on passing mechanisms, see the SQL/XML specification.

Operator results and combining with other operators
The result of the XMLEXISTS operator is a SQL boolean value that is based on the results from evaluating the `xquery-string-literal` against the `xml-value-expression`. The XMLEXISTS operator returns:

UNKNOWN
When the `xml-value-expression` is null.

TRUE
When the evaluation of the specified query expression against the specified xml-value returns a non-empty sequence of nodes or values.

FALSE
When evaluation of the specified query expression against the specified xml-value returns an empty sequence.

The XMLEXISTS operator does not return the actual results from the evaluation of the query. You must use the XMLQUERY operator to retrieve the actual results.

Since the result of the XMLEXISTS operator is an SQL boolean data type, you can use the XMLEXISTS operator wherever a boolean function is allowed. For example, you can use the XMLEXISTS operator as a check constraint in a table declaration or as a predicate in a WHERE clause.

Examples
In the `x_table` table, to determine if the `xcol` XML column for each row has an element called `student` with an `age` attribute equal to 20, use this statement:

```
SELECT id, XMLEXISTS('//student[@age=20]' PASSING BY REF xcol)
    FROM x_table
```

In the `x_table` table, to return the ID for every row whose `xcol` XML column is non-null and contains the element `/roster/student`, use this statement:

```
SELECT id FROM x_table WHERE XMLEXISTS('/roster/student' PASSING BY REF
 xcol)
```

You can create the `x_table` table with a check constraint that limits which XML values can be inserted into the `xcol` XML column. In this example, the constraint is that the column has at least one `student` element with an `age` attribute with a value that is less than 25. To create the table, use this statement:

```
CREATE TABLE x_table ( id INT, xcol XML CHECK (XMLEXISTS ('//student[@age
 < 25]' PASSING BY REF xcol)) )
```

Usage note

Derby requires that a JAXP parser (such as Apache Xerces) and Apache Xalan are listed in the Java classpath for the XML functions to work. If either the JAXP parser or Xalan is missing from the classpath, attempts to use the XMLEXISTS operator will result in an error. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

XMLPARSE operator

XMLPARSE is a SQL/XML operator that you use to parse a character string expression into a Derby XML value.

You can use the result of this operator temporarily or you can store the result permanently in Derby XML columns. Whether temporary or permanent, you can use the XML value as an input to the other Derby XML operators, such as XMLEXISTS and XMLQUERY.

Syntax

```
XMLPARSE (DOCUMENT string-value-expression PRESERVE WHITESPACE)
```

DOCUMENT

Required keyword that describes the type of XML input that Derby can parse. Derby can only parse string expressions that constitute well-formed XML documents. This is because Derby uses a JAXP parser to parse all string values. The JAXP parser expects the string-value-expression to constitute a well-formed XML document. If the string does not constitute a well-formed document, JAXP throws an error. Derby catches the error and throws the error as a SQLException.

`string-value-expression`

Any expression that evaluates to a SQL character type, such as CHAR, VARCHAR, LONG VARCHAR, or CLOB. The `string-value-expression` argument can also be a parameter. You must use the CAST function when you specify the parameter to indicate the type of value that is bound into the parameter. Derby must verify that the parameter is the correct data type before the value is parsed as an XML document. If a parameter is specified without the CAST function, or if the CAST is to a non-character datatype, Derby throws an error.

PRESERVE WHITESPACE

Required keywords that describe how Derby handles whitespace between consecutive XML nodes. When the PRESERVE WHITESPACE keywords are

used, Derby preserves whitespace as dictated by the SQL/XML rules for preserving whitespace.

For more information on what constitutes a well-formed XML document, see the following specification: http://www.w3.org/TR/REC-xml/#sec-well-formed .

Restriction: The SQL/XML standard dictates that the argument to the XMLPARSE operator can also be a binary string. However, Derby only supports character string input for the XMLPARSE operator.

Examples

To insert a simple XML document into the `xcol` XML column in the `x_table` table, use the following statement:

```
INSERT INTO x_table VALUES
    (1,
    XMLPARSE(DOCUMENT '
        <roster>
          <student age="18">AB</student>
          <student age="23">BC</student>
          <student>NOAGE</student>
        </roster>'
      PRESERVE WHITESPACE)
    )
```

To insert a large XML document into the `xcol` XML column in the `x_table` table, from JDBC use the following statement:

```
INSERT INTO x_table VALUES
    (2,
    XMLPARSE (DOCUMENT CAST (? AS CLOB) PRESERVE WHITESPACE)
    )
```

You should bind into the statement using the setCharacterStream() method, or any other JDBC setXXX method that works for the CAST target type.

Usage note

Derby requires that a JAXP parser (such as Apache Xerces) and Apache Xalan are listed in the Java classpath for the XML functions to work. If either the JAXP parser or Xalan is missing from the classpath, attempts to use the XMLPARSE operator will result in an error. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

XMLQUERY operator

XMLQUERY is a SQL/XML operator that you can use to query XML values in SQL.

The XMLQUERY operator has two arguments, an XML query expression and a Derby XML value.

Syntax

```
XMLQUERY ( xquery-string-literal
    PASSING BY REF xml-value-expression
    [ RETURNING SEQUENCE [ BY REF ] ]
    EMPTY ON EMPTY
    )
```

`xquery-string-literal`
Must be specified as a string literal. If this argument is specified as a parameter, an expression that is not a literal, or a literal that is not a string (for example an integer),Derby throws an error. The `xquery-string-literal` argument must

also be an XPath expression that is supported by Apache Xalan. Derby uses Apache Xalan to evaluate all XML query expressions. Because Xalan does not support full XQuery, neither does Derby. If Xalan is unable to compile or execute the query argument, Derby catches the error that is thrown by Xalan and throws the error as a SQLException. For more on XPath and XQuery expressions, see these Web sites: http://www.w3.org/TR/xpath and http://www.w3.org/TR/xquery/.

`xml-value-expression`
: Must be an XML data value and must constitute a well-formed SQL/XML document. The `xml-value-expression` argument cannot be a parameter. Derby does not perform implicit parsing nor casting of XML values, so use of strings or any other data type results in an error. If the argument is a sequence that is returned by a Derby XMLQUERY operation, the argument is accepted if it is a sequence of exactly one node that is a document node. Otherwise Derby throws an error.

BY REF
: Optional keywords that describe the only value passing mechanism supported by Derby. Since BY REF is also the default passing mechanism, the XMLQUERY operator behaves the same whether the keywords are present or not. For more information on passing mechanisms, see the SQL/XML specification.

RETURNING SEQUENCE
: Optional keywords that describe the only XML type returned by the Derby XMLQUERY operator. Since SEQUENCE is also the default return type, the XMLQUERY operator behaves the same whether the keywords are present or not. For more information on the different XML return types, see the SQL/XML specification.

EMPTY ON EMPTY
: Required keywords that describe the way in which XMLQUERY handles an empty result sequence. The XMLQUERY operator returns an empty sequence exactly as the sequence is. The XMLQUERY operator does not convert the empty sequence to a null value. When an empty result sequence is serialized, the result is an empty string. Derby does not consider an empty result sequence to be a well-formed XML document.

The result of the XMLQUERY operator is a value of type XML. The result represents a sequence of XML nodes or values. Atomic values, such as strings, can be part of the result sequence. The result of an XMLQUERY operator is not guaranteed to represent a well-formed XML document and it might not be possible to insert the result of an XMLQUERY operator into an XML column. To store the result in an XML column, the result must be a sequence with exactly one item in the sequence and the item must be a well-formed document node. The result can be viewed only in serialized form by explicitly using the XMLSERIALIZE operator.

Examples

In the `x_table` table, to search the XML column `xcol` and return the students that have an age attribute that is greater than 20, use the following statement:

```
SELECT ID,
    XMLSERIALIZE(
        XMLQUERY('//student[@age>20]' PASSING BY REF xcol EMPTY ON EMPTY)
    AS VARCHAR(50))
FROM x_table
```

The result set for this query contains a row for every row in `x_table`, regardless of whether or not the XMLQUERY operator actually returns results.

In the `x_table` table, to search the XML column `xcol` and return the ages for any students named BC, use the following statement:

```
SELECT ID,
```

```
    XMLSERIALIZE(
        XMLQUERY('string(//student[text() = "BC"]/@age)' PASSING BY REF
 xcol EMPTY ON EMPTY)
    AS VARCHAR(50))
FROM x_table
WHERE
    XMLEXISTS('//student[text() = "BC"]' PASSING BY REF xcol)
```

The result set for this query contains a row for only the rows in `x_table` that have a student whose name is BC.

Usage note

Derby requires that a JAXP parser (such as Apache Xerces) and Apache Xalan are listed in the Java classpath for the XML functions to work. If either the JAXP parser or Xalan is missing from the classpath, attempts to use the XMLQUERY operator will result in an error. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

XMLSERIALIZE operator

XMLSERIALIZE is a SQL/XML operator that you can use to convert an XML type to a character type. There is no other way to convert the type of a Derby XML value.

Attention: Serialization is performed based on the SQL/XML serialization rules. These rules, combined with the fact that Derby supports only a subset of the XMLSERIALIZE syntax, dictate that the results of an XMLSERIALIZE operation are not guaranteed to be in-tact copies of the original XML text. For example, assume that `[xString]` is a textual representation of a well-formed XML document. You issue the following statements:

```
INSERT INTO x_table (id, xcol)
    VALUES (3, XMLPARSE(DOCUMENT '[xString]' PRESERVE WHITESPACE));

SELECT id, XMLSERIALIZE(xcol AS VARCHAR(100))
    FROM x_table WHERE id = 3;
```

There is no guarantee that the result of the XMLSERIALIZE operator will be identical to the original `[xString]` representation. Certain transformations can occur as part of XMLSERIALIZE processing, and those transformations are defined in the SQL/XML specification. In some cases the result of XMLSERIALIZE might actually be the same as the original textual representation, but that is not guaranteed.

When an XMLSERIALIZE operator is specified as part of the top-level result set for a query, the result can be accessed from JDBC by using whatever JDBC getXXX methods are allowed on the `string-data-type` argument that is included in the XMLSERIALIZE syntax. If you attempt to select the contents of an XML value from a top-level result set without using the XMLSERIALIZE operator, Derby throws an error. Derby does not implicitly serialize XML values.

Syntax

```
XMLSERIALIZE ( xml-value-expression AS string-data-type )
```

`xml-value-expression`
: Can be any Derby XML value, including an XML result sequence generated by the XMLQUERY operator. The `xml-value-expression` argument cannot be a parameter.

`string-data-type`

Must be a SQL character string type, such as CHAR, VARCHAR, LONG VARCHAR, or CLOB. If you specify a type that is not a valid character string type, Derby throws an error.

Examples

In the `x_table` table, to display the contents of the `xcol` XML column, use this statement:

```
SELECT ID,
    XMLSERIALIZE(
    xcol AS CLOB)
FROM x_table
```

To retrieve the results from JDBC, you can use the JDBC getCharacterStream() or getString() method.

To display the results of an XMLQUERY operation, use the following statement:

```
SELECT ID,
    XMLSERIALIZE(
        XMLQUERY('//student[@age>20]'
           PASSING BY REF xcol EMPTY ON EMPTY)
    AS VARCHAR(50))
FROM x_table
```

Usage note

Derby requires that a JAXP parser (such as Apache Xerces) and Apache Xalan are listed in the Java classpath for the XML functions to work. If either the JAXP parser or Xalan is missing from the classpath, attempts to use the XMLSERIALIZE operator will result in an error. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

YEAR function

The YEAR function returns the year part of a value. The argument must be a date, timestamp, or a valid character string representation of a date or timestamp. The result of the function is an integer between 1 and 9 999. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Syntax

```
YEAR ( expression )
```

Example

Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

```
SELECT * FROM PROJECT WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
```

Built-in system functions

This section describes the different built-in system functions available with Derby.

SYSCS_UTIL.SYSCS_CHECK_TABLE system function

The `SYSCS_UTIL.SYSCS_CHECK_TABLE` function checks the specified table, ensuring that all of its indexes are consistent with the base table. If the table and indexes are

consistent, the method returns a SMALLINT with value 1. If the table and indexes are inconsistent, the function will throw an exception.

Syntax

```
SMALLINT SYSCS_UTIL.SYSCS_CHECK_TABLE(IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128))
```

An error will occur if either SCHEMANAME or TABLENAME are null.

Examples

Check a single table:

```
VALUES SYSCS_UTIL.SYSCS_CHECK_TABLE('SALES', 'ORDERS');
```

Check all tables:

```
SELECT schemaname, tablename, SYSCS_UTIL.SYSCS_CHECK_TABLE(schemaname,
tablename) FROM sys.sysschemas s, sys.systables t WHERE s.schemaid =
t.schemaid;
```

SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY system function

The `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function fetches the value of the specified property of the database on the current connection.

If the value that was set for the property is invalid, the `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function returns the invalid value, but Derby uses the default value.

Syntax

```
VARCHAR(32762) SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(IN KEY
 VARCHAR(128))
```

An error will be returned if KEY is null.

SQL example

Retrieve the value of the `derby.locks.deadlockTimeout` property:

```
VALUES
 SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY('derby.locks.deadlockTimeout');
```

SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS system function

The `SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS` function returns a VARCHAR(32762) value representing the query execution plan and run time statistics for a java.sql.ResultSet. A query execution plan is a tree of execution nodes. There are a number of possible node types. Statistics are accumulated during execution at each node. The types of statistics include the amount of time spent in specific operations, the number of rows passed to the node by its children, and the number of rows returned by the node to its parent. (The exact statistics are specific to each node type.) `SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS` is most meaningful for DML statements such as SELECT, INSERT, DELETE and UPDATE.

Syntax

```
VARCHAR(32762) SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

Example

```
VALUES SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()
```

SYSCS_UTIL.SYSCS_GET_USER_ACCESS system function

The SYSCS_UTIL.SYSCS_GET_USER_ACCESS function returns the current connection access permission for the user specified.

If no permission is explicitly set for the user, the access permission for the user is the value of the default connection mode. The default connection mode is set by using the derby.database.defaultConnectionMode property.

Syntax

```
SYSCS_UTIL.SYSCS_GET_USER_ACCESS (USERNAME VARCHAR(128)) RETURNS
 VARCHAR(128)
```

USERNAME
An input argument of type VARCHAR(128) that specifies the user ID in the Derby database.

The value that is returned by this function is either fullAccess, readOnlyAccess, or noAccess.

A return value of noAccess means that the connection attempt by the user will be denied because neither the derby.database.fullAccessUsers property nor the derby.database.readOnlyAccessUsers property is set for the user, and the derby.database.defaultConnectionMode property is set to noAccess.

The names of the connection permissions match the existing names in use by Derby.

Example

```
VALUES SYSCS_UTIL.SYSCS_GET_USER_ACCESS ('BRUNNER')
```

Built-in system procedures

Some built-in procedures are not compatible with SQL syntax used by other relational databases. These procedures can only be used with Derby.

SYSCS_UTIL.SYSCS_BACKUP_DATABASE system procedure

The SYSCS_UTIL.SYSCS_BACKUP_DATABASE system procedure backs up the database to a specified backup directory.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE(IN BACKUPDIR VARCHAR())
```

No result is returned from the procedure.

BACKUPDIR
An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, user.dir, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path.

JDBC example

The following example backs up the database to the c:/backupdir directory:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE(?)");
cs.setString(1, "c:/backupdir");
```

```
cs.execute();
cs.close();
```

SQL example

The following example backs up the database to the `c:/backupdir` directory:

```
CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE('c:/backupdir');
```

SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT system procedure

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT` system procedure backs up the database to a specified backup directory.

If there are any transactions in progress with unlogged operations at the start of the backup, the `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT` system procedure returns an error immediately, instead of waiting for those transactions to complete.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT(IN BACKUPDIR VARCHAR())
```

No result is returned from the procedure.

BACKUPDIR
An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path.

JDBC example

The following example backs up the database to the `c:/backupdir` directory:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT(?)");
cs.setString(1, "c:/backupdir");
cs.execute();
cs.close();
```

SQL example

The following example backs up the database to the `c:/backupdir` directory:

```
CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE_NOWAIT('c:/backupdir');
```

SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE system procedure

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE` system procedure backs up the database to a specified backup directory and enables the database for log archive mode.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE
(IN BACKUPDIR VARCHAR(32672), IN SMALLINT DELETE_ARCHIVED_LOG_FILES)
```

No result is returned from the procedure.

BACKUPDIR
An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current

user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path

DELETE_ARCHIVED_LOG_FILES

If the input parameter value for the DELETE_ARCHIVED_LOG_FILES parameter is a non-zero value, online archived log files that were created before this backup will be deleted. The log files are deleted only after a successful backup.

JDBC example

The following example backs up the database to the `c:/backupdir` directory:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE(?,
 ?)");
cs.setString(1, "c:/backupdir");
cs.setInt(2, 0);
cs.execute();
```

SQL examples

The following example backs up the database to the `c:/backupdir` directory, enables log archive mode, and does not delete any existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE('c:/
backupdir', 0)
```

The following example backs up the database to the `c:/backupdir` directory and, if this backup is successful, deletes existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE('c:/
backupdir', 1)
```

SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT system procedure

The `SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT` system procedure backs up the database to a specified backup directory and enables the database for log archive mode. This procedure returns an error if there are any transactions in progress that have unlogged operations at the start of the backup, instead of waiting for those transactions to complete.

Syntax

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT
(IN BACKUPDIR VARCHAR(32672),
IN SMALLINT DELETE_ARCHIVED_LOG_FILES)
```

No result is returned from the procedure.

BACKUPDIR

An input argument of type VARCHAR(32672) that specifies the path to a directory, where the backup should be stored. Relative paths are resolved based on the current user directory, `user.dir`, of the JVM where the database backup is occurring. Relative paths are not resolved based on the derby home directory. To avoid confusion, use the absolute path.

DELETE_ARCHIVED_LOG_FILES

If the input parameter value for the DELETE_ARCHIVED_LOG_FILES parameter is a non-zero value, online archived log files that were created before this backup will be deleted. The log files are deleted only after a successful backup.

JDBC example

The following example backs up the database to the `c:/backupdir` directory and enables log archive mode:

```
CallableStatement cs = conn.prepareCall
("CALL
 SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT(?,
 ?)");
cs.setString(1, "c:/backupdir");
cs.setInt(2, 0);
cs.execute();
```

SQL examples

The following example backs up the database to the `c:/backupdir` directory, enables log archive mode, and does not delete any existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT('c:/
backupdir', 0)
```

The following example backs up the database to the `c:/backupdir` directory and, if this backup is successful, deletes existing online archived log files:

```
SYSCS_UTIL.SYSCS_BACKUP_DATABASE_AND_ENABLE_LOG_ARCHIVE_MODE_NOWAIT('c:/
backupdir', 1)
```

SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE system procedure

The `SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE` stored procedure removes as many compiled statements (plans) as possible from the database-wide statement cache. The procedure does not remove statements related to currently executing queries or to activations that are about to be garbage collected, so the cache is not guaranteed to be completely empty after a call to this procedure.

Syntax

```
SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE()
```

JDBC Example

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE()");
cs.execute();
cs.close();
```

SQL Example

```
CALL SYSCS_UTIL.SYSCS_EMPTY_STATEMENT_CACHE();
```

SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE system procedure

The `SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE` system procedure checkpoints the database by flushing all cached data to disk.

Syntax

```
SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE()
```

No result is returned by this procedure.

JDBC example

```
CallableStatement cs = conn.prepareCall
```

```
("CALL SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE()");
cs.execute();
cs.close();
```

SQL Example

```
CALL SYSCS_UTIL.SYSCS_CHECKPOINT_DATABASE();
```

SYSCS_UTIL.SYSCS_COMPRESS_TABLE system procedure

Use the SYSCS_UTIL.SYSCS_COMPRESS_TABLE system procedure to reclaim unused, allocated space in a table and its indexes. Typically, unused allocated space exists when a large amount of data is deleted from a table, or indexes are updated. By default, Derby does not return unused space to the operating system. For example, once a page has been allocated to a table or index, it is not automatically returned to the operating system until the table or index is destroyed. SYSCS_UTIL.SYSCS_COMPRESS_TABLE allows you to return unused space to the operating system.

The SYSCS_UTIL.SYSCS_COMPRESS_TABLE system procedure updates statistics on all indexes as part of the index rebuilding process.

Syntax

```
SYSCS_UTIL.SYSCS_COMPRESS_TABLE (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN SEQUENTIAL SMALLINT)
```

SCHEMANAME
: An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a null will result in an error.

TABLENAME
: An input argument of type VARCHAR(128) that specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier 'Fred'. Passing a null will result in an error.

SEQUENTIAL
: A non-zero input argument of type SMALLINT will force the operation to run in sequential mode, while an argument of 0 will force the operation not to run in sequential mode. Passing a null will result in an error.

SQL example

To compress a table called CUSTOMER in a schema called US, using the SEQUENTIAL option:

```
call SYSCS_UTIL.SYSCS_COMPRESS_TABLE('US', 'CUSTOMER', 1)
```

Java example

To compress a table called CUSTOMER in a schema called US, using the SEQUENTIAL option:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_COMPRESS_TABLE(?, ?, ?)");
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 1);
cs.execute();
```

If the SEQUENTIAL parameter is not specified, Derby rebuilds all indexes concurrently with the base table. If you do not specify the SEQUENTIAL argument, this procedure can be memory-intensive and use a lot of temporary disk space (an amount equal to approximately two times the used space plus the unused, allocated space). This is

because Derby compresses the table by copying active rows to newly allocated space (as opposed to shuffling and truncating the existing space). The extra space used is returned to the operating system on COMMIT.

When SEQUENTIAL is specified, Derby compresses the base table and then compresses each index sequentially. Using SEQUENTIAL uses less memory and disk space, but is more time-intensive. Use the SEQUENTIAL argument to reduce memory and disk space usage.

`SYSCS_UTIL.SYSCS_COMPRESS_TABLE` cannot release any permanent disk space back to the operating system until a COMMIT is issued. This means that the space occupied by both the base table and its indexes cannot be released. Only the disk space that is temporarily claimed by an external sort can be returned to the operating system prior to a COMMIT.

Tip: We recommend that you issue the `SYSCS_UTIL.SYSCS_COMPRESS_TABLE` system procedure in the auto-commit mode.

Note: This procedure acquires an exclusive table lock on the table being compressed. All statement plans dependent on the table or its indexes are invalidated. For information on identifying unused space, see the *Java DB Server and Administration Guide*.

SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE system procedure

Use the `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` system procedure to reclaim unused, allocated space in a table and its indexes. Typically, unused allocated space exists when a large amount of data is deleted from a table and there has not been any subsequent inserts to use the space created by the deletes. By default, Derby does not return unused space to the operating system. For example, once a page has been allocated to a table or index, it is not automatically returned to the operating system until the table or index is destroyed. `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` allows you to return unused space to the operating system.

This system procedure can be used to force three levels of in-place compression of a SQL table: `PURGE_ROWS`, `DEFRAGMENT_ROWS`, and `TRUNCATE_END`. Unlike `SYSCS_UTIL.SYSCS_COMPRESS_TABLE()`, all work is done in place in the existing table/index.

Syntax

```
SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE(
  IN SCHEMANAME VARCHAR(128),
  IN TABLENAME VARCHAR(128),
  IN PURGE_ROWS SMALLINT,
  IN DEFRAGMENT_ROWS SMALLINT,
  IN TRUNCATE_END SMALLINT )
```

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a null will result in an error.

TABLENAME
An input argument of type VARCHAR(128) that specifies the table name of the table. The string must exactly match the case of the table name, and the argument of "Fred" will be passed to SQL as the delimited identifier 'Fred'. Passing a null will result in an error.

PURGE_ROWS
If PURGE_ROWS is set to a non-zero value, then a single pass is made through the table which will purge committed deleted rows from the table. This space is then available for future inserted rows, but remains allocated to the table. As this option scans every page of the table, its performance is linearly related to the size of the table.

DEFRAGMENT_ROWS

If DEFRAGMENT_ROWS is set to a non-zero value, then a single defragment pass is made which will move existing rows from the end of the table towards the front of the table. The goal of defragmentation is to empty a set of pages at the end of the table which can then be returned to the operating system by the TRUNCATE_END option. It is recommended to only run DEFRAGMENT_ROWS if also specifying the TRUNCATE_END option. The DEFRAGMENT_ROWS option scans the whole table and needs to update index entries for every base table row move, so the execution time is linearly related to the size of the table.

TRUNCATE_END

If TRUNCATE_END is set to a non-zero value, then all contiguous pages at the end of the table will be returned to the operating system. Running the PURGE_ROWS and/or DEFRAGMENT_ROWS options may increase the number of pages affected. This option by itself performs no scans of the table.

SQL example

To compress a table called CUSTOMER in a schema called US, using all available compress options:

```
call SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE('US', 'CUSTOMER', 1, 1, 1);
```

To return the empty free space at the end of the same table, the following call will run much quicker than running all options but will likely return much less space:

```
call SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE('US', 'CUSTOMER', 0, 0, 1);
```

Java example

To compress a table called CUSTOMER in a schema called US, using all available compress options:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE(?, ?, ?, ?, ?)");
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 1);
cs.setShort(4, (short) 1);
cs.setShort(5, (short) 1);
cs.execute();
```

To return the empty free space at the end of the same table, the following call will run much quicker than running all options but will likely return much less space:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE(?, ?, ?, ?, ?)");
cs.setString(1, "US");
cs.setString(2, "CUSTOMER");
cs.setShort(3, (short) 0);
cs.setShort(4, (short) 0);
cs.setShort(5, (short) 1);
cs.execute();
```

Tip: We recommend that you issue the `SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE` system procedure in the auto-commit mode.

Note: This procedure acquires an exclusive table lock on the table being compressed. All statement plans dependent on the table or its indexes are invalidated. For information on identifying unused space, see the *Java DB Server and Administration Guide*.

SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE system procedure

The SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE system procedure disables the log archive mode and deletes any existing online archived log files if the DELETE_ARCHIVED_LOG_FILES input parameter is non-zero.

Syntax

```
SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE(IN SMALLINT
 DELETE_ARCHIVED_LOG_FILES)
```

No result is returned from the procedure.

DELETE_ARCHIVED_LOG_FILES
If the input parameter value for the DELETE_ARCHIVED_LOG_FILES parameter is a non-zero value, then all existing online archived log files are deleted. If the parameter value is zero, then exiting online archived log files are not deleted.

JDBC example

The following example disables log archive mode for the database and deletes any existing log archive files.

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE(?)");
cs.setInt(1, 1);
cs.execute();
cs.close();
```

SQL examples

The following example disables log archive mode for the database and retains any existing log archive files:

```
CALL SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE
 DELETE_ARCHIVED_LOG_FILES(0);
```

The following example disables log archive mode for the database and deletes any existing log archive files:

```
CALL SYSCS_UTIL.SYSCS_DISABLE_LOG_ARCHIVE_MODE
 DELETE_ARCHIVED_LOG_FILES(1);
```

SYSCS_UTIL.SYSCS_EXPORT_TABLE system procedure

The SYSCS_UTIL.SYSCS_EXPORT_TABLE system procedure exports all of the data from a table to an operating system file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_TABLE (IN SCHEMANAME  VARCHAR(128),
IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128))
```

No result is returned from the procedure.

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema name of the table. Passing a NULL value will use the default schema name.

TABLENAME
An input argument of type VARCHAR(128) that specifies the name of the table/view from which the data is to be exported. Passing a null will result in an error.

FILENAME
Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the exported file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the database code set to the specified code set before writing to the file. Passing a NULL value will write the data in the same code set as the JVM in which it is being executed.

If you create a schema or table name as a non-delimited identifier, you must pass the name to the export procedure using all uppercase characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the export procedure using the same case that was used when it was created.

Usage

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example

The following example shows how to export information from the STAFF table in a SAMPLE database to the `myfile.del` file.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE (null, 'STAFF', 'myfile.del', null,
 null, null);
```

SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE` system procedure to export all the data from a table, and place the LOB data into a separate export file. A reference to the location of the LOB data is placed in the LOB column in the main export file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE (
```

```
    IN SCHEMANAME VARCHAR(128),
    IN TABLENAME VARCHAR(128),
    IN FILENAME VARCHAR(32672),
    IN COLUMNDELIMITER CHAR(1),
    IN CHARACTERDELIMITER CHAR(1),
    IN CODESET VARCHAR(128)
    IN LOBSFILENAME VARCHAR(32672)
    )
```

When you run this procedure, the column data is written to the main export file in a delimited data file format.

SCHEMANAME
: Specifies the schema of the table. You can specify a NULL value to use the default schema name. The SCHEMANAME parameter takes an input argument that is a VARCHAR (128) data type.

TABLENAME
: Specifies the table name of the table or view from which the data is to be exported. This table cannot be a system table or a declared temporary table. The string must exactly match the case of the table name. Specifying a NULL value results in an error. The TABLENAME parameter takes an input argument that is a VARCHAR (128) data type.

FILENAME
: Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
: Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter must be a CHAR (1) data type.

CHARACTERDELIMITER
: Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET
: Specifies the code set of the data in the export file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the database code page to the specified code page before writing to the file. You can specify a NULL value to write the data in the same code page as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

LOBSFILENAME
: Specifies the file that the large object data is exported to. If the path is omitted, the lob file is created in the same directory as the main export file. If you specify the name of an existing file, the export utility overwrites the contents of the file. The data is not appended to the file. If you are using the Network Server, the file should be in a server-side location. Specifying a NULL value results in an error. The LOBSFILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the export procedure using all uppercase characters. If you created a schema or table name as a delimited identifier, you must pass the name to the export procedure using the same case that was used when it was created.

Usage

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example exporting all data from a table, using a separate export file for the LOB data

The following example shows how to export data from the STAFF table in a sample database to the main file `staff.del` and the LOB export file `pictures.dat`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE_LOBS_TO_EXTFILE(
    'APP', 'STAFF', 'c:\data\staff.del', ',' ,'"',
    'UTF-8', 'c:\data\pictures.dat');
```

SYSCS_UTIL.SYSCS_EXPORT_QUERY system procedure

The `SYSCS_UTIL.SYSCS_EXPORT_QUERY` system procedure exports the results of a SELECT statement to an operating system file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_QUERY(IN SELECTSTATEMENT VARCHAR(32672),
IN FILENAME VARCHAR(32672), IN COLUMNDELIMITER CHAR(1),
IN CHARACTERDELIMITER CHAR(1), IN CODESET VARCHAR(128))
```

No result is returned from the procedure.

SELECTSTATEMENT
An input argument of type VARCHAR(32672) that specifies the select statement (query) that will return the data to be exported. Passing a NULL value will result in an error.

FILENAME
Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER
An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET
An input argument of type VARCHAR(128) that specifies the code set of the data in the exported file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the database code set to the specified code set before writing to the file. Passing a NULL value will write the data in the same code set as the JVM in which it is being executed.

Usage

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example

The following example shows how to export the information about employees in Department 20 from the STAFF table in the SAMPLE database to the `myfile.del` file.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY('select * from staff where dept =20',
    'c:/output/awards.del', null, null, null);
```

SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE` system procedure to export the result of a SELECT statement to a main export file, and place the LOB data into a separate export file. A reference to the location of the LOB data is placed in the LOB column in the main export file.

For security concerns, and to avoid accidental file damage, this EXPORT procedure does not export data into an existing file. You must specify a filename in the EXPORT procedure that does not exist. When you run the procedure the file is created and the data is exported into the new file.

The data is exported using a delimited file format.

Syntax

```
SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE (
    IN SELECTSTATEMENT VARCHAR(32672),
    IN FILENAME VARCHAR(32672),
    IN COLUMNDELIMITER CHAR(1),
    IN CHARACTERDELIMITER CHAR(1),
    IN CODESET VARCHAR(128)
    IN LOBSFILENAME VARCHAR(32672)
    )
```

When you run this procedure, the column data is written to the main export file in a delimited data file format.

SELECTSTATEMENT
Specifies the SELECT statement query that returns the data to be exported. Specifying a NULL value will result in an error. The SELECTSTATEMENT parameter takes an input argument that is a VARCHAR (32672) data type.

FILENAME
Specifies the name of a new file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export procedure returns an error. The specified location of the file should refer to the server-side location if you are using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter must be a CHAR (1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET

Specifies the code set of the data in the export file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the database code page to the specified code page before writing to the file. You can specify a NULL value to write the data in the same code page as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

LOBSFILENAME

Specifies the file that the large object data is exported to. If the path is omitted, the lob file is created in the same directory as the main export file. If you specify the name of an existing file, the export utility overwrites the contents of the file. The data is not appended to the file. If you are using the Network Server, the file should be in a server-side location. Specifying a NULL value results in an error. The LOBSFILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

Usage

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example exporting data from a query using a separate export file for the LOB data

The following example shows how to export employee data in department 20 from the STAFF table in a sample database to the main file `staff.del` and the lob data to the file `pictures.dat`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY_LOBS_TO_EXTFILE(
    'SELECT * FROM STAFF WHERE dept=20',
    'c:\data\staff.del', ',' ,'"',
    'UTF-8','c:\data\pictures.dat');
```

SYSCS_UTIL.SYSCS_IMPORT_DATA system procedure

The `SYSCS_UTIL.SYSCS_IMPORT_DATA` system procedure imports data to a subset of columns in a table. You choose the subset of columns by specifying insert columns. This procedure is also used to import a subset of column data from a file by specifying column indexes.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_DATA (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN INSERTCOLUMNS VARCHAR(32672),
IN COLUMNINDEXES VARCHAR(32672), IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128), IN REPLACE SMALLINT)
```

No result is returned from the procedure.

SCHEMANAME

An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a NULL value will use the default schema name.

TABLENAME

An input argument of type VARCHAR (128) that specifies the table name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. Passing a null will result in an error.

INSERTCOLUMNS

An input argument of type VARCHAR (32762) that specifies the column names (separated by commas) of the table into which the data is to be imported. Passing a NULL value will import the data into all of the columns of the table.

COLUMNINDEXES

An input argument of type VARCHAR (32762) that specifies the indexes (numbered from 1 and separated by commas) of the input data fields to be imported. Passing a NULL value will use all of the input data fields in the file.

FILENAME

An input argument of type VARCHAR(32672) that specifies the file that contains the data to be imported. If you do not specify a path, the current working directory is used. Passing a NULL value will result in an error.

COLUMNDELIMITER

An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER

An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET

An input argument of type VARCHAR(128) that specifies the code set of the data in the input file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the specified code set to the database code set (utf-8). Passing a NULL value will interpret the data file in the same code set as the JVM in which it is being executed.

REPLACE

A input argument of type SMALLINT. A non-zero value will run in REPLACE mode, while a value of zero will run in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the data object, and inserts the imported data. The table definition and the index definitions are not changed. You can only use the REPLACE mode if the table exists. INSERT mode adds the imported data to the table without changing the existing table data. Passing a NULL will result in an error.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all uppercase characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Usage

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example

The following example imports some of the data fields from a delimited data file called `data.del` into the STAFF table:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA
    (NULL, 'STAFF', null, '1,3,4', 'data.del', null, null, null,0)
```

SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE` system procedure to import data to a subset of columns in a table, where the LOB data is stored in a separate file. The main import file contains all of the other data and a reference to the location of the LOB data.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE (
   IN SCHEMANAME VARCHAR(128),
   IN TABLENAME VARCHAR(128),
```

```
   IN INSERTCOLUMNS VARCHAR(32672),
   IN COLUMNINDEXES VARCHAR(32672),
   IN FILENAME VARCHAR(32672),
   IN COLUMNDELIMITER CHAR(1),
   IN CHARACTERDELIMITER CHAR(1),
   IN CODESET VARCHAR(128),
   IN REPLACE SMALLINT)
   )
```

The import utility looks in the main import file for a reference to the location of the LOB data.

SCHEMANAME
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The SCHEMANAME parameter takes an input argument that is a VARCHAR (128) data type.

TABLENAME
Specifies the name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. The string must exactly match case of the table name. Specifying a NULL value results in an error. The TABLENAME parameter takes an input argument that is a VARCHAR (128) data type.

INSERTCOLUMNS
Specifies the comma separated column names of the table into which the data will be imported. You can specify a NULL value to import into all columns of the table. The INSERTCOLUMNS parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNINDEXES
Specifies the comma separated column indexes (numbered from one) of the input data fields that will be imported. You can specify a NULL value to use all input data fields in the file. The COLUMNINDEXES parameter takes an input argument that is a VARCHAR (32762) data type.

FILENAME
Specifies the name of the file that contains the data to be imported. If the path is omitted, the current working directory is used. The specified location of the file should refer to the server side location if using the Network Server. Specifying a NULL value results in an error. The fileName parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET
Specifies the code set of the data in the input file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the specified code set to the database code set (UTF-8). You can specify a NULL value to interpret the data file in the same code set as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

REPLACE
A non-zero value for the replace parameter will import in REPLACE mode, while a zero value will import in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the table and inserts the imported data. The table

definition and the index definitions are not changed. You can only import with REPLACE mode if the table already exists. INSERT mode adds the imported data to the table without changing the existing table data. Specifying a NULL value results in an error. The REPLACE parameter takes an input argument that is a SMALLINT data type.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all uppercase characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Usage

This procedure will read the LOB data using the reference that is stored in the main import file. The format of the reference to the LOB stored in the main import file must be `lobsFileName.Offset.length/`.

- `Offset` is position in the external file in bytes
- `length` is the size of the LOB column data in bytes

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example importing data into specific columns, using a separate import file for the LOB data

The following example shows how to import data into several columns of the STAFF table. The STAFF table includes a LOB column in a sample database. The import file `staff.del` is a delimited data file. The `staff.del` file contains references that point to a separate file which contains the LOB data. The data in the import file is formatted using double quotation marks (") as the string delimiter and a comma (,) as the column delimiter. The data will be appended to the existing data in the STAFF table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA_LOBS_FROM_EXTFILE
    (null, 'STAFF', 'NAME,DEPT,SALARY,PICTURE', '2,3,4,6',
    'c:\data\staff.del', ',','"','UTF-8',  0);
```

SYSCS_UTIL.SYSCS_IMPORT_TABLE system procedure

The `SYSCS_UTIL.SYSCS_IMPORT_TABLE` system procedure imports data from an input file into all of the columns of a table. If the table receiving the imported data already contains data, you can either replace or append to the existing data.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_TABLE (IN SCHEMANAME VARCHAR(128),
IN TABLENAME VARCHAR(128), IN FILENAME VARCHAR(32672),
IN COLUMNDELIMITER CHAR(1), IN CHARACTERDELIMITER CHAR(1),
IN CODESET VARCHAR(128), IN REPLACE SMALLINT)
```

No result is returned from the procedure.

SCHEMANAME
An input argument of type VARCHAR(128) that specifies the schema of the table. Passing a NULL value will use the default schema name.

TABLENAME
An input argument of type VARCHAR (128) that specifies the table name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. Passing a null will result in an error.

FILENAME

An input argument of type VARCHAR(32672) that specifies the file that contains the data to be imported. If you do not specify a path, the current working directory is used. Passing a NULL value will result in an error.

COLUMNDELIMITER

An input argument of type CHAR(1) that specifies a column delimiter. The specified character is used in place of a comma to signal the end of a column. Passing a NULL value will use the default value; the default value is a comma (,).

CHARACTERDELIMITER

An input argument of type CHAR(1) that specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. Passing a NULL value will use the default value; the default value is a double quotation mark (").

CODESET

An input argument of type VARCHAR(128) that specifies the code set of the data in the input file. The name of the code set should be one of the Java-supported character encodings. Data is converted from the specified code set to the database code set (utf-8). Passing a NULL value will interpret the data file in the same code set as the JVM in which it is being executed.

REPLACE

A input argument of type SMALLINT. A non-zero value will run in REPLACE mode, while a value of zero will run in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the data object, and inserts the imported data. The table definition and the index definitions are not changed. INSERT mode adds the imported data to the table without changing the existing table data. Passing a NULL will result in an error.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all uppercase characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Usage

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example

The following example imports data into the `STAFF` table from a delimited data file called `myfile.del` with the percentage character (%) as the string delimiter, and a semicolon (;) as the column delimiter:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
    (null, 'STAFF', 'c:/output/myfile.del', ';', '%', null,0);
```

SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE system procedure

Use the `SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE` system procedure to import data to a table, where the LOB data is stored in a separate file. The main import file contains all of the other data and a reference to the location of the LOB data.

Syntax

```
SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE (
   IN SCHEMANAME VARCHAR(128),
   IN TABLENAME VARCHAR(128),
   IN FILENAME VARCHAR(32672),
   IN COLUMNDELIMITER CHAR(1),
   IN CHARACTERDELIMITER CHAR(1),
   IN CODESET VARCHAR(128),
```

```
    IN REPLACE SMALLINT)
    )
```

The import utility looks in the main import file for a reference to the location of the LOB data.

SCHEMANAME
Specifies the schema of the table. You can specify a NULL value to use the default schema name. The SCHEMANAME parameter takes an input argument that is a VARCHAR (128) data type.

TABLENAME
Specifies the name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. The string must exactly match case of the table name. Specifying a NULL value results in an error. The TABLENAME parameter takes an input argument that is a VARCHAR (128) data type.

FILENAME
Specifies the name of the file that contains the data to be imported. If the path is omitted, the current working directory is used. The specified location of the file should refer to the server side location if using the Network Server. Specifying a NULL value results in an error. The FILENAME parameter takes an input argument that is a VARCHAR (32672) data type.

COLUMNDELIMITER
Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can specify a NULL value to use the default value of a comma. The COLUMNDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CHARACTERDELIMITER
Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can specify a NULL value to use the default value of a double quotation mark. The CHARACTERDELIMITER parameter takes an input argument that is a CHAR (1) data type.

CODESET
Specifies the code set of the data in the input file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the specified code set to the database code set (UTF-8). You can specify a NULL value to interpret the data file in the same code set as the JVM in which it is being executed. The CODESET parameter takes an input argument that is a VARCHAR (128) data type.

REPLACE
A non-zero value for the replace parameter will import in REPLACE mode, while a zero value will import in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the table and inserts the imported data. The table definition and the index definitions are not changed. You can only import with REPLACE mode if the table already exists. INSERT mode adds the imported data to the table without changing the existing table data. Specifying a NULL value results in an error. The REPLACE parameter takes an input argument that is a SMALLINT data type.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all uppercase characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Usage

This procedure will read the LOB data using the reference that is stored in the main import file. If you are importing from a non-Derby source, the format of the reference to the LOB stored in the main import file must be `lobsFileName.Offset.length/`.

- `Offset` is position in the external file in bytes
- `length` is the size of the LOB column data in bytes

For additional information on using this procedure see the section "Using the bulk import and export procedures" in the *Java DB Tools and Utilities Guide*.

Example importing data from a main import file that contains references which point to a separate file that contains LOB data

The following example shows how to import data into the *STAFF* table in a sample database from a delimited data file `staff.del`. This example defines a comma as the column delimiter. The data will be appended to the existing data in the table.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE_LOBS_FROM_EXTFILE(
    'APP','STAFF','c:\data\staff.del',',','"','UTF-8',0);
```

SYSCS_UTIL.SYSCS_FREEZE_DATABASE system procedure

The `SYSCS_UTIL.SYSCS_FREEZE_DATABASE` system procedure temporarily freezes the database for backup.

Syntax

```
SYSCS_UTIL.SYSCS_FREEZE_DATABASE()
```

No result set is returned by this procedure.

Example

```
String backupdirectory = "c:/mybackups/" + JCalendar.getToday();
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
cs.execute();
cs.close();
// user supplied code to take full backup of "backupdirectory"
// now unfreeze the database once backup has completed:
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
cs.execute();
cs.close();
```

SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE system procedure

The `SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE` system procedure unfreezes a database after backup.

Syntax

```
SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()
```

No result set is returned by this procedure.

Example

```
String backupdirectory = "c:/mybackups/" + JCalendar.getToday();
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()");
cs.execute();
cs.close();
// user supplied code to take full backup of "backupdirectory"
// now unfreeze the database once backup has completed:
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()");
cs.execute();
cs.close();
```

SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY system procedure

The SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY system procedure reloads the security policy, allowing you to fine-tune your Java security on the fly. For more information on security policies, see the section titled "Running the Network Server under the security manager" in the *Java DB Server and Administration Guide* and the section titled "Running Derby under a security manager" in the *Java DB Developer's Guide*.

Syntax

```
SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY()
```

No result set is returned by this procedure.

Example

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY()");
cs.execute();
cs.close();
```

SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure

Use the SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY system procedure to set or delete the value of a property of the database on the current connection. For information about properties, see Derby property reference.

If VALUE is not null, then the property with key value KEY is set to VALUE. If VALUE is null, then the property with key value KEY is deleted from the database property set.

If VALUE is an invalid value for the property, Derby uses the default value of the property, although SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY returns the invalid value.

Syntax

```
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(IN KEY VARCHAR(128),
IN VALUE VARCHAR(32672))
```

This procedure does not return any results.

JDBC example

Set the derby.locks.deadlockTimeout property to a value of 10:

```
CallableStatement cs = conn.prepareCall
("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(?, ?)");
cs.setString(1, "derby.locks.deadlockTimeout");
cs.setString(2, "10");
cs.execute();
cs.close();
```

SQL example

Set the derby.locks.deadlockTimeout property to a value of 10:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
('derby.locks.deadlockTimeout', '10')
```

SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS system procedure

The SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS() system procedure turns a connection's runtime statistics on or off. By default, the runtime statistics are turned off.

When the `runtimestatistics` attribute is turned on, Derby maintains information about the execution plan for each statement executed within the connection (except for COMMIT) until the attribute is turned off. To turn the `runtimestatistics` attribute off, call the procedure with an argument of zero. To turn the `runtimestatistics` on, call the procedure with any non-zero argument.

For statements that do not return rows, the object is created when all internal processing has completed before returning to the client program. For statements that return rows, the object is created when the first `next()` call returns 0 rows or if a `close()` call is encountered, whichever comes first.

Syntax

```
SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(IN SMALLINT ENABLE)
```

Example

```
-- establish a connection
-- turn on RUNTIMESTATISTIC for connection:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
-- execute complex query here
-- step through the result sets
-- access runtime statistics information:
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(0);
```

SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING system procedure

Statistics timing is an attribute associated with a connection that you turn on and off by using the `SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING` system procedure. Statistics timing is turned off by default. Turn statistics timing on only when the `runtimestatistics` attribute is already on. Turning statistics timing on when the `runtimestatistics` attribute is off has no effect.

Turn statistics timing on by calling this procedure with a non-zero argument. Turn statistics timing off by calling the procedure with a zero argument.

When statistics timing is turned on, Derby tracks the timings of various aspects of the execution of a statement. This information is included in the information returned by the SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS system function. When statistics timing is turned off, the SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS system function shows all timing values as zero.

Syntax

```
SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(IN SMALLINT ENABLE)
```

Example

To turn the `runtimestatistics` attribute and then the statistics timing attribute on:

```
CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);
```

SYSCS_UTIL.SYSCS_SET_USER_ACCESS system procedure

The `SYSCS_UTIL.SYSCS_SET_USER_ACCESS` system procedure sets the connection access permission for the user specified.

Syntax

```
SYSCS_UTIL.SYSCS_SET_USER_ACCESS (USERNAME VARCHAR(128),
 CONNECTION_PERMISSION VARCHAR(128))
```

USERNAME

An input argument of type VARCHAR(128) that specifies the user ID in the Derby database.

CONNECTION_PERMISSION

Valid values for CONNECTION_PERMISSION are:

fullAccess

Adds the user to the list of users with full access to the database. The value for the database property is `derby.database.fullAccessUsers`.

readOnlyAccess

Adds the user to the list of users with read-only access to the database. The value for the database property is `derby.database.readOnlyAccessUsers`.

null

Removes the user from the list of permissions, reverting the user to the default permission. You must specify `null` without the quotation marks.

Example

```
CALL SYSCS_UTIL.SYSCS_SET_USER_ACCESS ('BRUNNER', 'readOnlyAccess')
```

To remove the user from the list of permissions, you specify the `null` value without the quotation marks. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_USER_ACCESS ('ISABEL', null)
```

SYSCS_UTIL.SYSCS_UPDATE_STATISTICS system procedure

The SYSCS_UTIL.SYSCS_UPDATE_STATISTICS system procedure updates the cardinality statistics, or creates the statistics if they do not exist, for the index that you specify or for all of the indexes on a table. Derby uses cardinality statistics to determine the optimal query plan during the compilation of a query. If the statistics are missing, Derby might use a query plan which is not the most efficient plan.

Once statistics have been created, they cannot be dropped and should be maintained. It is a good idea to call the SYSCS_UTIL.SYSCS_UPDATE_STATISTICS procedure when the number of distinct values in an index is likely to have changed significantly.

For more information on cardinality statistics, see "Working with cardinality statistics" in the *Tuning Java DB* guide.

Syntax

```
SYSCS_UTIL.SYSCS_UPDATE_STATISTICS(IN SCHEMANAME VARCHAR(128),
                                   IN TABLENAME VARCHAR(128),
                                   IN INDEXNAME VARCHAR(128)))
```

Note: You can specify `null` for the INDEXNAME to update any existing statistics and create statistics for those statistics that are missing.

Examples

In the following example, the system procedure updates statistics for the index PAY_DESC on the SAMP.EMPLOYEE table:

```
CALL SYSCS_UTIL.SYSCS_UPDATE_STATISTICS('SAMP','EMPLOYEE','PAY_DESC');
```

In the following example, `null` is specified instead of an index name. For all of the indexes, the existing statistics are updated and statistics are created for any missing statistics on the EMPLOYEE table in the SAMP schema.

```
CALL SYSCS_UTIL.SYSCS_UPDATE_STATISTICS('SAMP', 'EMPLOYEE', null);
```

SYSCS_DIAG diagnostic tables and functions

Derby provides a set of system table expressions which you can use to obtain diagnostic information about the state of the database and about the database sessions.

There are two types of diagnostic table expressions in Derby:

Diagnostic tables
: Tables that are like any other table in Derby. You can specify the diagnostic table name anywhere a normal table name is allowed.

Diagnostic table functions
: Functions that are like any other function in Derby. Diagnostic table functions can accept zero or more arguments, depending on the table function that you use. You must use the SQL-defined table function syntax to access these functions.

The following table shows the types and names of the diagnostic table expressions in Derby.

Table 9. System diagnostic table expressions provided by Derby

Diagnostic table expression	Type of expression
SYSCS_DIAG.CONTAINED_ROLES	Table function
SYSCS_DIAG.ERROR_LOG_READER	Table function
SYSCS_DIAG.ERROR_MESSAGES	Table
SYSCS_DIAG.LOCK_TABLE	Table
SYSCS_DIAG.SPACE_TABLE	Table function
SYSCS_DIAG.STATEMENT_CACHE	Table
SYSCS_DIAG.STATEMENT_DURATION	Table function
SYSCS_DIAG.TRANSACTION_TABLE	Table

Restriction: If you reference a diagnostic table in a DDL statement or a compression procedure, Derby returns an exception.

SYSCS_DIAG.CONTAINED_ROLES diagnostic table function

The SYSCS_DIAG.CONTAINED_ROLES diagnostic table function returns all the roles contained within the specified role. For a definition of role containment, see "Syntax for roles" in GRANT statement.

For example:

```
SELECT * FROM TABLE (SYSCS_DIAG.CONTAINED_ROLES(reader))
```

SYSCS_DIAG.ERROR_LOG_READER diagnostic table function

The SYSCS_DIAG.ERROR_LOG_READER diagnostic table function contains all the useful SQL statements that are in the `derby.log` file or a log file that you specify.

One use of this diagnostic table function is to determine the active transactions and the SQL statements in those transactions at a given point in time. For example, if a deadlock or lock timeout occurred you can find the timestamp (timestampConstant) in the error log.

To access the SYSCS_DIAG.ERROR_LOG_READER diagnostic table function, you must use the SQL table function syntax.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.ERROR_LOG_READER())
    AS T1
```

where T1 is a user-specified table name that is any valid identifier.

You can specify a log file name as an optional argument to the SYSCS_DIAG.ERROR_LOG_READER diagnostic table function. When you specify a log file name, the file name must be an expression whose data type maps to a Java string.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.ERROR_LOG_READER('myderbyerrors.log'))
    AS T1
```

Tip: By default Derby log files contain only boot, shutdown, and error messages. See the `derby.stream.error.logSeverityLevel` property and the `derby.language.logStatementText` property for instructions on how to print more information to Derby log files. You can then query that information by using the SYSCS_DIAG.ERROR_LOG_READER diagnostic table function.

SYSCS_DIAG.ERROR_MESSAGES diagnostic table

The SYSCS_DIAG.ERROR_MESSAGES diagnostic table shows all of the SQLStates, locale-sensitive error messages, and exception severities for a Derby database. You can reference the SYSCS_DIAG.ERROR_MESSAGES diagnostic table directly in a statement.

For example:

```
SELECT * FROM SYSCS_DIAG.ERROR_MESSAGES
```

SYSCS_DIAG.LOCK_TABLE diagnostic table

The SYSCS_DIAG.LOCK_TABLE diagnostic table shows all of the locks that are currently held in the Derby database. You can reference the SYSCS_DIAG.LOCK_TABLE diagnostic table directly in a statement.

For example:

```
SELECT * FROM SYSCS_DIAG.LOCK_TABLE
```

When the SYSCS_DIAG.LOCK_TABLE diagnostic table is referenced in a statement, a snap shot of the lock table is taken. A snap shot is used so that referencing the diagnostic table does not alter the normal timing and flow of the application. It is possible that some locks will be in a transition state when the snap shot is taken.

SYSCS_DIAG.SPACE_TABLE diagnostic table function

The SYSCS_DIAG.SPACE_TABLE diagnostic table function shows the space usage of a particular table and its indexes. You can use this diagnostic table function to determine if space might be saved by compressing the table and indexes.

To access the SYSCS_DIAG.SPACE_TABLE diagnostic table function, you must use the SQL table function syntax. This diagnostic table function takes two arguments, the schemaName and the tableName. The tableName argument is required. If you do not specify the schemaName, the current schema is used.

For example, use the following query to return the space usage for all of the user tables and indexes in the database:

```
SELECT T2.*
    FROM
        SYS.SYSTABLES systabs,
```

```
        TABLE (SYSCS_DIAG.SPACE_TABLE(systabs.tablename)) AS T2
    WHERE systabs.tabletype = 'T'
```

where T2 is a user-specified table name that is any valid identifier.

Both the schemaName and the tableName arguments must be expressions whose data types map to Java strings. If the schemaName and the tableName are non-delimited identifiers, you must specify the names in upper case.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.SPACE_TABLE('MYSCHEMA', 'MYTABLE'))
    AS T2
```

SYSCS_DIAG.STATEMENT_CACHE diagnostic table

The SYSCS_DIAG.STATEMENT_CACHE diagnostic table shows the contents of the SQL statement cache. You can reference the SYSCS_DIAG.STATEMENT_CACHE diagnostic table directly in a statement.

For example:

```
SELECT * FROM SYSCS_DIAG.STATEMENT_CACHE
```

SYSCS_DIAG.STATEMENT_DURATION diagnostic table function

You can use the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function to analyze the execution duration of the useful SQL statements in the `derby.log` file or a log file that you specify.

You can also use this diagnostic table function to get an indication of where the bottlenecks are in the JDBC code for an application.

To access the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function, you must use the SQL table function syntax.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.STATEMENT_DURATION())
    AS T1
```

where T1 is a user-specified table name that is any valid identifier.

Restriction: For each transaction ID, a row is not returned for the last statement with that transaction id. Transaction IDs change within a connection after a commit or rollback, if the transaction that just ended modified data.

You can specify a log file name as an optional argument to the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function. When you specify a log file name, the file name must be an expression whose data type maps to a Java string.

For example:

```
SELECT *
    FROM TABLE (SYSCS_DIAG.STATEMENT_DURATION('somederby.log'))
    AS T1
```

Tip: By default Derby log files contain only boot, shutdown, and error messages. See the `derby.stream.error.logSeverityLevel` property and the `derby.language.logStatementText` property for instructions on how to print more information to Derby log files. You can then query that information by using the SYSCS_DIAG.STATEMENT_DURATION diagnostic table function.

SYSCS_DIAG.TRANSACTION_TABLE diagnostic table

The SYSCS_DIAG.TRANSACTION_TABLE diagnostic table shows all of the transactions that are currently in the database. You can reference the SYSCS_DIAG.TRANSACTION_TABLE diagnostic table directly in a statement.

For example:

```
SELECT * FROM SYSCS_DIAG.TRANSACTION_TABLE
```

When the SYSCS_DIAG.TRANSACTION_TABLE diagnostic table is referenced in a statement, a snap shot of the transaction table is taken. A snap shot is used so that referencing the diagnostic table does not alter the normal timing and flow of the application. It is possible that some transactions will be in a transition state when the snap shot is taken.

Data types

This section describes the data types used in Derby.

Built-In type overview

The SQL type system is used by the language compiler to determine the compile-time type of an expression and by the language execution system to determine the runtime type of an expression, which can be a subtype or implementation of the compile-time type.

Each type has associated with it values of that type. In addition, values in the database or resulting from expressions can be NULL, which means the value is missing or unknown. Although there are some places where the keyword NULL can be explicitly used, it is not in itself a value, because it needs to have a type associated with it.

The syntax presented in this section is the syntax you use when specifying a column's data type in a CREATE TABLE statement.

Numeric types

Numeric types used in Derby.

Numeric type overview

Numeric types include the following types, which provide storage of varying sizes:

- Integer numerics
  - SMALLINT (2 bytes)
  - INTEGER (4 bytes)
  - BIGINT (8 bytes)
- Approximate or floating-point numerics
  - REAL (4 bytes)
  - DOUBLE PRECISION (8 bytes)
  - FLOAT (an alias for DOUBLE PRECISION or REAL)
- Exact numeric
  - DECIMAL (storage based on precision)
  - NUMERIC (an alias for DECIMAL)

Numeric type promotion in expressions

In expressions that use only integer types, Derby promotes the type of the result to at least INTEGER. In expressions that mix integer with non-integer types, Derby promotes the result of the expression to the highest type in the expression. Type Promotion in Expressions shows the promotion of data types in expressions.

Table 10. Type Promotion in Expressions

Largest Type That Appears in Expression	Resulting Type of Expression
DOUBLE PRECISION	DOUBLE PRECISION
REAL	DOUBLE PRECISION
DECIMAL	DECIMAL
BIGINT	BIGINT
INTEGER	INTEGER
SMALLINT	INTEGER

For example:

```
-- returns a double precision
VALUES 1 + 1.0e0
-- returns a decimal
VALUES 1 + 1.0
-- returns an integer
VALUES CAST (1 AS INT) + CAST (1 AS INT)
```

Storing values of one numeric data type in columns of another numeric data type

An attempt to put a floating-point type of a larger storage size into a location of a smaller size fails only if the value cannot be stored in the smaller-size location. For example:

```
create table mytable (r REAL, d DOUBLE PRECISION);
0 rows inserted/updated/deleted
INSERT INTO mytable (r, d) values (3.4028236E38, 3.4028235E38);
ERROR X0X41: The number '3.4028236E38' is outside the range for
the data type REAL.
```

You can store a floating point type in an INTEGER column; the fractional part of the number is truncated. For example:

```
INSERT INTO mytable(integer_column) values (1.09e0);
1 row inserted/updated/deleted
SELECT integer_column
FROM mytable;
I
---------------
1
```

Integer types can always be placed successfully in approximate numeric values, although with the possible loss of some precision.

Integers can be stored in decimals if the DECIMAL precision is large enough for the value. For example:

```
ij> insert into mytable (decimal_column)
VALUES (55555555556666666666);
ERROR X0Y21: The number '55555555556666666666' is outside the
range of the target DECIMAL/NUMERIC(5,2) datatype.
```

An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. For example:

```
INSERT INTO mytable (int_column) values 2147483648;
ERROR 22003: The resulting value is outside the range for the
data type INTEGER.
```

Note: When truncating trailing digits from a NUMERIC value, Derby rounds down.

Scale for decimal arithmetic

SQL statements can involve arithmetic expressions that use decimal data types of different *precision*s (the total number of digits, both to the left and to the right of the decimal point) and *scale*s (the number of digits of the fractional component). The precision and scale of the resulting decimal type depend on the precision and scale of the operands.

Given an arithmetic expression that involves two decimal operands:

- *lp* stands for the precision of the left operand
- *rp* stands for the precision of the right operand
- *ls* stands for the scale of the left operand
- *rs* stands for the scale of the right operand

Use the following formulas to determine the scale of the resulting data type for the following kinds of arithmetical expressions:

- *multiplication*

  ls + rs
- *division*

  31 - lp + ls - rs
- *AVG()*

  max(max(ls, rs), 4)
- *all others*

  max(ls, rs)

For example, the scale of the resulting data type of the following expression is 27:

```
11.0/1111.33
// 31 - 3 + 1 - 2 = 27
```

Use the following formulas to determine the precision of the resulting data type for the following kinds of arithmetical expressions:

- *multiplication*

  lp + rp
- *addition*

  2 * (p - s) + s
- *division*

  lp - ls + rp + max(ls + rp - rs + 1, 4)
- *all others*

  max(lp - ls, rp - rs) + 1 + max(ls, rs)

Data type assignments and comparison, sorting, and ordering

Table 11. Assignments allowed by Derby

This table displays valid assignments between data types in Derby. A "Y" indicates that the assignment is valid.

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP	XML
SMALL INT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
FLOAT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
CHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y	-
VARCHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	Y	Y	Y	-
LONG VARCHAR	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-
CHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-	-
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-	-
LONG VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	-	-	-
CLOB	-	-	-	-	-	-	-	Y	Y	Y	-	-	-	Y	-	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP	XML
DATE	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-
TIME STAMP	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

Table 12. Comparisons allowed by Derby
This table displays valid comparisons between data types in Derby. A "Y" indicates that the comparison is allowed.

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP	XML
SMALL INT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
FLOAT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
CHAR	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y	-
VARCHAR	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y	-
LONG VARCHAR	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-
VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-
LONG VARCHAR FOR BIT DATA	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Types	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	FLOAT	CHAR	VARCHAR	LONG VARCHAR	CHAR FOR BIT DATA	VARCHAR FOR BIT DATA	LONG VARCHAR FOR BIT DATA	CLOB	BLOB	DATE	TIME	TIMESTAMP	XML
DATE	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-
TIME STAMP	-	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	-	Y	-
XML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

BIGINT data type

BIGINT provides 8 bytes of storage for integer values.

Syntax

```
BIGINT
```

Corresponding compile-time Java type

java.lang.Long

JDBC metadata type (java.sql.Types)

BIGINT

Minimum value

-9223372036854775808 (*java.lang.Long.MIN_VALUE*)

Maximum value

9223372036854775807 (*java.lang.Long.MAX_VALUE*)

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions.

An attempt to put an integer value of a larger storage size into a location of a smaller size fails if the value cannot be stored in the smaller-size location. Integer types can always successfully be placed in approximate numeric values, although with the possible loss of some precision. BIGINTs can be stored in DECIMALs if the DECIMAL precision is large enough for the value.

```
9223372036854775807
```

BLOB data type

A BLOB (binary large object) is a varying-length binary string that can be up to 2,147,483,647 characters long. Like other binary types, BLOB strings are not associated with a code page. In addition, BLOB strings do not hold character data.

The length is given in bytes for BLOB unless one of the suffixes K, M, or G is given, relating to the multiples of 1024, 1024*1024, 1024*1024*1024 respectively.

Note: Length is specified in bytes for BLOB.

Syntax

```
{ BLOB | BINARY LARGE OBJECT } [ ( length [{K |M |G }] ) ]
```

Default

A BLOB without a specified length is defaulted to two gigabytes (2,147,483,647).

Corresponding compile-time Java type

java.sql.Blob

JDBC metadata type (java.sql.Types)

BLOB

Use the *getBlob* method on the *java.sql.ResultSet* to retrieve a BLOB handle to the underlying data.

Related information

see Mapping of java.sql.Blob and java.sql.Clob interfaces

```
create table pictures(name varchar(32) not null primary key, pic
 blob(16M));

--find all logotype pictures
select length(pic), name from pictures where name like '%logo%';

--find all image doubles (blob comparsions)
select a.name as double_one, b.name as double_two
from pictures as a, pictures as b
where a.name < b.name
and a.pic = b.pic
order by 1,2;
```

CHAR data type

CHAR provides for fixed-length storage of strings.

Syntax

```
CHAR[ACTER] [(length)]
```

length is an unsigned integer literal designating the length in bytes. The default *length* for a CHAR is 1, and the maximum size of *length* is 254.

Corresponding compile-time Java type

java.lang.String

JDBC metadata type (java.sql.Types)

CHAR

Derby inserts spaces to pad a string value shorter than the expected length. Derby truncates spaces from a string value longer than the expected length. Characters other than spaces cause an exception to be raised. When comparison boolean operators are applied to CHARs, the shorter string is padded with spaces to the length of the longer string.

When CHARs and VARCHARs are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.

The type of a string constant is CHAR.

Examples

```
-- within a string constant use two single quotation marks
-- to represent a single quotation mark or apostrophe
VALUES 'hello this is Joe''s string'
```

```
-- create a table with a CHAR field
CREATE TABLE STATUS (
    STATUSCODE CHAR(2) NOT NULL
        CONSTRAINT PK_STATUS PRIMARY KEY,
    STATUSDESC VARCHAR(40) NOT NULL
);
```

CHAR FOR BIT DATA data type

A CHAR FOR BIT DATA type allows you to store byte strings of a specified length. It is useful for unstructured data where character strings are not appropriate.

Syntax

```
{ CHAR | CHARACTER }[(length)] FOR BIT DATA
```

length is an unsigned integer literal designating the length in bytes.

The default *length* for a CHAR FOR BIT DATA type is 1., and the maximum size of *length* is 254 bytes.

JDBC metadata type (java.sql.Types)

BINARY

CHAR FOR BIT DATA stores fixed-length byte strings. If a CHAR FOR BIT DATA value is smaller than the target CHAR FOR BIT DATA, it is padded with a 0x20 byte value.

Comparisons of CHAR FOR BIT DATA and VARCHAR FOR BIT DATA values are precise. For two bit strings to be equal, they must be *exactly* the same length. (This differs from the way some other DBMSs handle BINARY values but works as specified in SQL-92.)

An operation on a VARCHAR FOR BIT DATA and a CHAR FOR BIT DATA value (e.g., a concatenation) yields a VARCHAR FOR BIT DATA value.

```
CREATE TABLE t (b CHAR(2) FOR BIT DATA);
INSERT INTO t VALUES (X'DE');
```

```
SELECT *
FROM t;
-- yields the following output
B
-----
de20
```

CLOB data type

A CLOB (character large object) value can be up to 2,147,483,647 characters long. A CLOB is used to store unicode character-based data, such as large documents in any character set.

The length is given in number characters for both CLOB, unless one of the suffixes K, M, or G is given, relating to the multiples of 1024, 1024*1024, 1024*1024*1024 respectively.

Length is specified in characters (unicode) for CLOB.

Syntax

```
{CLOB |CHARACTER LARGE OBJECT} [ ( length [{K |M |G}] ) ]
```

Default

A CLOB without a specified length is defaulted to two giga characters (2,147,483,647).

Corresponding Compile-Time Java Type

java.sql.Clob

JDBC Metadata Type (java.sql.Types)

CLOB

Use the *getClob* method on the *java.sql.ResultSet* to retrieve a CLOB handle to the underlying data.

Related Information

See Mapping of java.sql.Blob and java.sql.Clob interfaces.

```
import java.sql.*;

public class clob
{
    public static void main(String[] args) {
        try {
            String url = "jdbc:derby:clobberyclob;create=true";

            // Load the driver. This code is not needed if you are using
            // JDK 6, because in that environment the driver is loaded
            // automatically when the application requests a connection.
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

            Connection conn = DriverManager.getConnection(url);

            Statement s = conn.createStatement();
            s.executeUpdate(
                "CREATE TABLE documents (id INT, text CLOB(64 K))");
            conn.commit();

            // --- add a file
            java.io.File file = new java.io.File("asciifile.txt");
            int fileLength = (int) file.length();

            // - first, create an input stream
            java.io.InputStream fin = new java.io.FileInputStream(file);
            PreparedStatement ps = conn.prepareStatement("INSERT
```

```
            INTO documents VALUES (?, ?)");
            ps.setInt(1, 1477);

            // - set the value of the input parameter to the input stream
            ps.setAsciiStream(2, fin, fileLength);
            ps.execute();
            conn.commit();

            // --- reading the columns
            ResultSet rs = s.executeQuery(
                "SELECT text FROM documents WHERE id = 1477");
            while (rs.next()) {
                java.sql.Clob aclob = rs.getClob(1);
                java.io.InputStream ip = rs.getAsciiStream(1);
                int c = ip.read();
                while (c > 0) {
                    System.out.print((char)c);
                    c = ip.read();
                }
                System.out.print("\n");
                // ...
            }
        } catch (Exception e) {
            System.out.println("Error! "+e);
        }
    }
}
```

DATE data type

DATE provides for storage of a year-month-day in the range supported by *java.sql.Date*.

Syntax

```
DATE
```

Corresponding compile-time Java type

java.sql.Date

JDBC metadata type (java.sql.Types)

DATE

Dates, times, and timestamps must not be mixed with one another in expressions.

Any value that is recognized by the *java.sql.Date* method is permitted in a column of the corresponding SQL date/time data type. Derby supports the following formats for DATE:

```
yyyy-mm-dd
mm/dd/yyyy
dd.mm.yyyy
```

The first of the three formats above is the *java.sql.Date* format.

The year must always be expressed with four digits, while months and days may have either one or two digits.

Derby also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Examples

```
VALUES DATE('1994-02-23')

VALUES '1993-09-01'
```

DECIMAL data type

DECIMAL provides an exact numeric in which the precision and scale can be arbitrarily sized. You can specify the precision (the total number of digits, both to the left and the right of the decimal point) and the scale (the number of digits of the fractional component). The amount of storage required is based on the precision.

Syntax

```
{ DECIMAL | DEC } [(precision [, scale ])]
```

The *precision* must be between 1 and 31. The *scale* must be less than or equal to the precision.

If the scale is not specified, the default scale is 0. If the precision is not specified, the default precision is 5.

An attempt to put a numeric value into a DECIMAL is allowed as long as any non-fractional precision is not lost. When truncating trailing digits from a DECIMAL value, Derby rounds down.

For example:

```
-- this cast loses only fractional precision
values cast (1.798765 AS decimal(5,2));
1
--------
1.79
-- this cast does not fit
values cast (1798765 AS decimal(5,2));
1
--------
ERROR 22003: The resulting value is outside the range
for the data type DECIMAL/NUMERIC(5,2).
```

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions.

See also Storing values of one numeric data type in columns of another numeric data type.

When two decimal values are mixed in an expression, the scale and precision of the resulting value follow the rules shown in Scale for decimal arithmetic.

Corresponding compile-time Java type

java.math.BigDecimal

JDBC metadata type (java.sql.Types)

DECIMAL

```
VALUES 123.456

VALUES 0.001
```

Integer constants too big for BIGINT are made DECIMAL constants.

DOUBLE data type

The DOUBLE data type is a synonym for the DOUBLE PRECISION data type.

Syntax

```
DOUBLE
```

DOUBLE PRECISION data type

The DOUBLE PRECISION data type provides 8-byte storage for numbers using IEEE floating-point notation.

Syntax

```
DOUBLE PRECISION
```

or, alternately

```
DOUBLE
```

DOUBLE can be used synonymously with DOUBLE PRECISION.

Limitations

DOUBLE value ranges:

- Smallest DOUBLE value: -1.79769E+308
- Largest DOUBLE value: 1.79769E+308
- Smallest positive DOUBLE value: 2.225E-307
- Largest negative DOUBLE value: -2.225E-307

These limits are different from the `java.lang.Double`Java type limits.

An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If the values are too small, you will receive an exception.

Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:
values 01234567890123456789012345678901e0;
```

Corresponding compile-time Java type

java.lang.Double

JDBC metadata type (java.sql.Types)

DOUBLE

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions.

See also Storing values of one numeric data type in columns of another numeric data type.

Examples

```
3421E+09
425.43E9
9E-10
4356267544.32333E+30
```

FLOAT data type

The FLOAT data type is an alias for a REAL or DOUBLE PRECISION data type, depending on the precision you specify.

Syntax

```
FLOAT [ (precision) ]
```

The default *precision* for FLOAT is 53 and is equivalent to DOUBLE PRECISION. A precision of 23 or less makes FLOAT equivalent to REAL. A precision of 24 or greater makes FLOAT equivalent to DOUBLE PRECISION. If you specify a precision of 0, you get an error. If you specify a negative precision, you get a syntax error.

JDBC metadata type (java.sql.Types)

REAL or DOUBLE

Limitations

If you are using a precision of 24 or greater, the limits of FLOAT are similar to the limits of DOUBLE.

If you are using a precision of 23 or less, the limits of FLOAT are similar to the limits of REAL.

INTEGER data type

INTEGER provides 4 bytes of storage for integer values.

Syntax

```
{ INTEGER | INT }
```

Corresponding Compile-Time Java Type

java.lang.Integer

JDBC Metadata Type (java.sql.Types)

INTEGER

Minimum Value

-2147483648 (*java.lang.Integer.MIN_VALUE*)

Maximum Value

2147483647 (*java.lang.Integer.MAX_VALUE*)

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions.

See also Storing values of one numeric data type in columns of another numeric data type.

```
3453
425
```

LONG VARCHAR data type

The LONG VARCHAR type allows storage of character strings with a maximum length of 32,700 characters. It is identical to VARCHAR, except that you do not have to specify a maximum length when creating columns of this type.

Syntax

```
LONG VARCHAR
```

Corresponding compile-time Java type

java.lang.String

JDBC metadata type (java.sql.Types)

LONGVARCHAR

When you are converting from Java values to SQL values, no Java type corresponds to LONG VARCHAR.

LONG VARCHAR FOR BIT DATA data type

The LONG VARCHAR FOR BIT DATA type allows storage of bit strings up to 32,700 bytes. It is identical to VARCHAR FOR BIT DATA, except that you do not have to specify a maximum length when creating columns of this type.

Syntax

```
LONG VARCHAR FOR BIT DATA
```

JDBC metadata type (java.sql.Types)

LONGVARBINARY

NUMERIC data type

NUMERIC is a synonym for DECIMAL and behaves the same way. See DECIMAL data type.

Syntax

```
NUMERIC [(precision [, scale ])]
```

Corresponding compile-time Java type

java.math.BigDecimal

JDBC metadata Ttype (java.sql.Types)

NUMERIC

```
123.456
.001
```

REAL data type

The REAL data type provides 4 bytes of storage for numbers using IEEE floating-point notation.

Syntax

```
REAL
```

Corresponding compile-time Java type

java.lang.Float

JDBC metadata type (java.sql.Types)

REAL

Limitations

REAL value ranges:

- Smallest REAL value: -3.402E+38
- Largest REAL value: 3.402E+38
- Smallest positive REAL value: 1.175E-37
- Largest negative REAL value: -1.175E-37

These limits are different from the `java.lang.Float` Java type limits.

An exception is thrown when any double value is calculated or entered that is outside of these value ranges. Arithmetic operations **do not** round their resulting values to zero. If the values are too small, you will receive an exception. The arithmetic operations take place with double arithmetic in order to detect under flows.

Numeric floating point constants are limited to a length of 30 characters.

```
-- this example will fail because the constant is too long:
values 01234567890123456789012345678901e0;
```

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions.

See also Storing values of one numeric data type in columns of another numeric data type.

Constants always map to DOUBLE PRECISION; use a CAST to convert a constant to a REAL.

SMALLINT data type

SMALLINT provides 2 bytes of storage.

Syntax

```
SMALLINT
```

Corresponding compile-time Java type

java.lang.Short

JDBC metadata type (java.sql.Types)

SMALLINT

Minimum value

-32768 (*java.lang.Short.MIN_VALUE*)

Maximum value

32767 (*java.lang.Short.MAX_VALUE*)

When mixed with other data types in expressions, the resulting data type follows the rules shown in Numeric type promotion in expressions.

See also Storing values of one numeric data type in columns of another numeric data type.

Constants in the appropriate format always map to INTEGER or BIGINT, depending on their length.

TIME data type

TIME provides for storage of a time-of-day value.

Syntax

```
TIME
```

Corresponding compile-time Java type

java.sql.Time

JDBC metadata type (java.sql.Types)

TIME

Dates, times, and timestamps cannot be mixed with one another in expressions except with a CAST.

Any value that is recognized by the *java.sql.Time* method is permitted in a column of the corresponding SQL date/time data type. Derby supports the following formats for TIME:

```
hh:mm[:ss]
hh.mm[.ss]
hh[:mm] {AM | PM}
```

The first of the three formats above is the *java.sql.Time* format.

Hours may have one or two digits. Minutes and seconds, if present, must have two digits.

Derby also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Examples

```
VALUES TIME('15:09:02')
VALUES '15:09:02'
```

TIMESTAMP data type

TIMESTAMP stores a combined DATE and TIME value to be stored. It permits a fractional-seconds value of up to nine digits.

Syntax

```
TIMESTAMP
```

Corresponding compile-time Java type

java.sql.Timestamp

JDBC metadata type (java.sql.Types)

TIMESTAMP

Dates, times, and timestamps cannot be mixed with one another in expressions.

Derby supports the following formats for TIMESTAMP:

```
yyyy-mm-dd hh:mm:ss[.nnnnnn]
yyyy-mm-dd-hh.mm.ss[.nnnnnn]
```

The first of the two formats above is the *java.sql.Timestamp* format.

The year must always have four digits. Months, days, and hours may have one or two digits. Minutes and seconds must have two digits. Nanoseconds, if present, may have between one and six digits.

Derby also accepts strings in the locale specific datetime format, using the locale of the database server. If there is an ambiguity, the built-in formats above take precedence.

Examples

```
VALUES '1960-01-01 23:03:20'
VALUES TIMESTAMP('1962-09-23 03:23:34.234')
VALUES TIMESTAMP('1960-01-01 23:03:20')
```

VARCHAR data type

VARCHAR provides for variable-length storage of strings.

Syntax

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING }(length)
```

length is an unsigned integer constant, and it must not be greater than the constraint of the integer used to specify the length, the value *java.lang.Integer.MAX_VALUE*.

The maximum length for a VARCHAR string is 32,672 characters.

Corresponding compile-time Java type

java.lang.String

JDBC metadata type (java.sql.Types)

VARCHAR

Derby does not pad a VARCHAR value whose length is less than specified. Derby truncates spaces from a string value when a length greater than the VARCHAR expected is provided. Characters other than spaces are not truncated, and instead cause an exception to be raised. When comparison boolean operators are applied to VARCHARs, the lengths of the operands are not altered, and spaces at the end of the values are ignored.

When CHARs and VARCHARs are mixed in expressions, the shorter value is padded with spaces to the length of the longer value.

The type of a string constant is CHAR, not VARCHAR.

VARCHAR FOR BIT DATA data type

The VARCHAR FOR BIT DATA type allows you to store binary strings less than or equal to a specified length. It is useful for unstructured data where character strings are not appropriate (e.g., images).

Syntax

```
{ VARCHAR | CHAR VARYING | CHARACTER VARYING } (length) FOR BIT DATA
```

length is an unsigned integer literal designating the length in bytes.

Unlike the case for the CHAR FOR BIT DATA type, there is no default *length* for a VARCHAR FOR BIT DATA type. The maximum size of the *length* value is 32,672 bytes.

JDBC metadata type (java.sql.Types)

VARBINARY

VARCHAR FOR BIT DATA stores variable-length byte strings. Unlike CHAR FOR BIT DATA values, VARCHAR FOR BIT DATA values are not padded out to the target length.

An operation on a VARCHAR FOR BIT DATA and a CHAR FOR BIT DATA value (e.g., a concatenation) yields a VARCHAR FOR BIT DATA value.

The type of a byte literal is always a VARCHAR FOR BIT DATA, not a CHAR FOR BIT DATA.

XML data type

The XML data type is used for Extensible Markup Language (XML) documents.

The XML data type is used:

- To store XML documents that conform to the SQL/XML definition of a well-formed XML(DOCUMENT(ANY)) value.
- Transiently for XML(SEQUENCE) values, that might not be well-formed XML(DOCUMENT(ANY)) values.

Note: For an application to retrieve, update, query, or otherwise access an XML data value, the application must have classes for a JAXP parser and for Xalan in the classpath. Derby issues an error if either the parser or Xalan is not found. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

Because none of the JDBC-side support for SQL/XML is implemented in Derby, it is not possible to bind directly into an XML value or to retrieve an XML value directly from a result set using JDBC. Instead, you must bind and retrieve the XML data as Java strings or character streams by explicitly specifying the appropriate XML operators, XMLPARSE and XMLSERIALIZE, as part of your SQL queries.

Syntax

```
XML
```

Corresponding compile-time Java type

None

The Java type for XML values is java.sql.SQLXML. However, the java.sql.SQLXML type is not supported by Derby.

JDBC metadata type (java.sql.Types)

None

The metadata type for XML values is SQLXML. However, the SQLXML type is not supported by Derby.

To retrieve XML values from a Derby database using JDBC, use the XMLSERIALIZE operator in the SQL query. For example:

```
SELECT XMLSERIALIZE (xcol as CLOB) FROM myXmlTable
```

Then retrieve the XML value by using the getXXX method that corresponds to the target serialization type, in this example CLOB data types.

To store an XML value into a Derby database using JDBC, use the XMLPARSE operator in the SQL statement. For example:

```
INSERT INTO myXmlTable(xcol) VALUES XMLPARSE(
    DOCUMENT CAST (? AS CLOB) PRESERVE WHITESPACE)
```

Then use any of the setXXX methods that are compatible with String types, in this example use the PreparedStatement.setString or PreparedStatement.setCharacterStream method calls to bind the operator.

Argument matching

When you declare a function or procedure using CREATE FUNCTION/PROCEDURE, Derby does not verify whether a matching Java method exists. Instead, Derby looks for a matching method only when you invoke the function or procedure in a later SQL statement. At that time, Derby searches for a public, static method having the class and method name declared in the EXTERNAL NAME clause of the earlier CREATE FUNCTION/PROCEDURE statement. Furthermore, the Java types of the method's arguments and return value must match the SQL types declared in the CREATE FUNCTION/PROCEDURE statement. The following may happen:

- **Success** - If exactly one Java method matches, then Derby invokes it.
- **Ambiguity** - Derby raises an error if more than one method matches.
- **Failure** - Derby also raises an error if no method matches.

In mapping SQL data types to Java data types, Derby considers the following kinds of matches:

- **Primitive match** - Derby looks for a primitive Java type corresponding to the SQL type. For instance, SQL INTEGER matches Java *int*.
- **Wrapper match** - Derby looks for a wrapper class in the *java.lang* or *java.sql* packages corresponding to the SQL type. For instance, SQL INTEGER matches *java.lang.Integer*.
- **Array match** - For OUT and INOUT procedure arguments, Derby looks for an array of the corresponding primitive or wrapper type. For instance, an OUT procedure argument of type SQL INTEGER matches *int[]* and *Integer[]*.
- **ResultSet match** - If a procedure is declared to return *n* RESULT SETS, then Derby looks for a method whose last *n* arguments are of type *java.sql.ResultSet[]*.

Derby resolves function and procedure invocations as follows:

- **Function** - Derby looks for a method whose argument and return types are *primitive match*es or *wrapper match*es for the function's SQL arguments and return value.
- **Procedure** - Derby looks for a method which returns void and whose argument types match as follows:
  - *IN* - Method arguments are *primitive match*es or *wrapper match*es for the procedure's IN arguments.
  - *OUT and INOUT* - Method arguments are *array match*es for the procedure's OUT and INOUT arguments.

  In addition, if the procedure returns *n* RESULT SETS, then the last *n* arguments of the Java method must be of type *java.sql.ResultSet[]*.

Derby provides a tool, `SignatureChecker`, which can identify any SQL functions or procedures in a database that do not follow these argument matching rules. See the *Java DB Tools and Utilities Guide* for details.

Example of argument matching

The following function...

```
CREATE FUNCTION TO_DEGREES
( RADIANS DOUBLE )
RETURNS DOUBLE
PARAMETER STYLE JAVA
NO SQL LANGUAGE JAVA
EXTERNAL NAME 'acme.MathUtils.toDegrees'
```

...would match all of the following methods:

```
public static double toDegrees( double arg ) {...}
public static Double toDegrees( double arg ) {...}
public static double toDegrees( Double arg ) {...}
public static Double toDegrees( Double arg ) {...}
```

Note that Derby would raise an exception if Derby found more than one matching method.

Mapping SQL data types to Java data types

The following table shows how Derby maps specific SQL data types to Java data types:

Table 13. SQL and Java type correspondence

SQL type	Primitive match	Wrapper match
SMALLINT	*short*	*java.lang.Integer*
INTEGER	*int*	*java.lang.Integer*
BIGINT	*long*	*java.lang.Long*
DECIMAL	-	*java.math.BigDecimal*
NUMERIC	-	*java.math.BigDecimal*
REAL	*float*	*java.lang.Float*
DOUBLE	*double*	*java.lang.Double*
FLOAT	*double*	*java.lang.Double*
CHAR	-	*java.lang.String*
VARCHAR	-	*java.lang.String*
LONG VARCHAR	-	*java.lang.String*
CHAR FOR BIT DATA	*byte[]*	-
VARCHAR FOR BIT DATA	*byte[]*	-
LONG VARCHAR FOR BIT DATA	*byte[]*	-
CLOB	-	*java.sql.Clob*
BLOB	-	*java.sql.Blob*
DATE	-	*java.sql.Date*
TIME	-	*java.sql.Time*
TIMESTAMP	-	*java.sql.Timestamp*
XML	-	-

SQL reserved words

This section lists all the Derby reserved words, including those in the SQL-92 standard. Derby will return an error if you use any of these keywords as an identifier name unless you surround the identifier name with quotes ("). See Rules for SQL92 identifiers.

ADD
ALL
ALLOCATE
ALTER
AND
ANY
ARE
AS
ASC
ASSERTION
AT
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIGINT
BIT
BOOLEAN
BOTH
BY
CALL
CASCADE
CASCADED
CASE
CAST
CHAR
CHARACTER
CHECK
CLOSE
COALESCE
COLLATE
COLLATION
COLUMN
COMMIT
CONNECT
CONNECTION
CONSTRAINT
CONSTRAINTS
CONTINUE
CONVERT
CORRESPONDING
CREATE
CURRENT
CURRENT_DATE
CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER

CURSOR
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFERRABLE
DEFERRED
DELETE
DESC
DESCRIBE
DIAGNOSTICS
DISCONNECT
DISTINCT
DOUBLE
DROP
ELSE
END
END-EXEC
ESCAPE
EXCEPT
EXCEPTION
EXEC
EXECUTE
EXISTS
EXPLAIN
EXTERNAL
FALSE
FETCH
FIRST
FLOAT
FOR
FOREIGN
FOUND
FROM
FULL
FUNCTION
GET
GETCURRENTCONNECTION
GLOBAL
GO
GOTO
GRANT
GROUP
HAVING
HOUR
IDENTITY
IMMEDIATE
IN
INDICATOR
INITIALLY
INNER
INOUT
INPUT
INSENSITIVE
INSERT

INT
INTEGER
INTERSECT
INTO
IS
ISOLATION
JOIN
KEY
LAST
LEFT
LIKE
LOWER
LTRIM
MATCH
MAX
MIN
MINUTE
NATIONAL
NATURAL
NCHAR
NVARCHAR
NEXT
NO
NONE
NOT
NULL
NULLIF
NUMERIC
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OUTER
OUTPUT
OVER
OVERLAPS
PAD
PARTIAL
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURE
PUBLIC
READ
REAL
REFERENCES
RELATIVE
RESTRICT
REVOKE
RIGHT
ROLLBACK

ROWS
ROW_NUMBER
RTRIM
SCHEMA
SCROLL
SECOND
SELECT
SESSION_USER
SET
SMALLINT
SOME
SPACE
SQL
SQLCODE
SQLERROR
SQLSTATE
SUBSTR
SUBSTRING
SUM
SYSTEM_USER
TABLE
TEMPORARY
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRANSACTION
TRANSLATE
TRANSLATION
TRIM
TRUE
UNION
UNIQUE
UNKNOWN
UPDATE
UPPER
USER
USING
VALUES
VARCHAR
VARYING
VIEW
WHENEVER
WHERE
WITH
WORK
WRITE
XML
XMLEXISTS
XMLPARSE
XMLQUERY
XMLSERIALIZE
YEAR

Derby support for SQL-92 features

There are four levels of SQL-92 support:

- SQL92E

  Entry
- SQL92T

  Transitional, a level defined by NIST in a publication called FIPS 127-2
- SQL92I

  Intermediate
- SQL92F

  Full

Basic data types

The following table shows Derby support for the SQL-92 basic data types.

Table 14. Support for SQL-92 Features: Basic data types

Feature	Source	Derby
SMALLINT	SQL92E	Yes
INTEGER	SQL92E	Yes
DECIMAL(p,s)	SQL92E	Yes
NUMERIC(p,s)	SQL92E	Yes
REAL	SQL92E	Yes
FLOAT(p)	SQL92E	Yes
DOUBLE PRECISION	SQL92E	Yes
CHAR(n)	SQL92E	Yes

Basic math operations

Table 15. Support for SQL-92 Features: Basic math operations

Feature	Source	Derby
+, *, -, /, unary +, unary -	SQL92E	Yes

Basic comparisons

Table 16. Support for SQL-92 Features: Basic comparisons

Feature	Source	Derby
<, >, <= ,>=, <>, =	SQL92E	Yes

Basic predicates

Table 17. Support for SQL-92 Features: Basic predicates

Feature	Source	Derby
BETWEEN, LIKE, NULL	SQL92E	Yes

Quantified predicates

Table 18. Support for SQL-92 Features: Quantified predicates

Feature	Source	Derby
IN, ALL/SOME, EXISTS	SQL92E	Yes

Schema definition

Table 19. Support for SQL-92 Features: schema definition

Feature	Source	Derby
Tables	SQL92E	Yes
Views	SQL92E	Yes
Privileges	SQL92E	Yes

Column attributes

Table 20. Support for SQL-92 Features: column attributes

Feature	Source	Derby
Default values	SQL92E	Yes
Nullability	SQL92E	Yes

Constraints (non-deferrable)

Table 21. Support for SQL-92 Features: constraints (non-deferrable)

Feature	Source	Derby
NOT NULL	SQL92E	Yes (not stored in SYSCONSTRAINTS)
UNIQUE/PRIMARY KEY	SQL92E	Yes
FOREIGN KEY	SQL92E	Yes
CHECK	SQL92E	Yes
View WITH CHECK OPTION	SQL92E	No, views cannot be updated

Cursors

Table 22. Support for SQL-92 Features: Cursors

Feature	Source	Derby
DECLARE, OPEN, FETCH, CLOSE	SQL92E	Yes, by using JDBC method calls
UPDATE, DELETE CURRENT	SQL92E	Yes

Dynamic SQL 1

Table 23. Support for SQL-92 Features: Dynamic SQL 1

Feature	Source	Derby
ALLOCATE / DEALLOCATE / GET / SET DESCRIPTOR	SQL92T	Yes, by using JDBC method calls
PREPARE / EXECUTE / EXECUTE IMMEDIATE	SQL92T	Yes, by using JDBC method calls
DECLARE, OPEN, FETCH, CLOSE, UPDATE, DELETE dynamic cursor	SQL92T	Yes, by using JDBC method calls
DESCRIBE output	SQL92T	Yes, by using JDBC method calls

Basic information schema

Table 24. Support for SQL-92 Features: Basic information schema

Feature	Source	Derby
TABLES	SQL92T	*SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS*
VIEWS	SQL92T	*SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS*
COLUMNS	SQL92T	*SYS.SYSTABLES, SYS.SYSVIEWS, SYS.SYSCOLUMNS*

Basic schema manipulation

Table 25. Support for SQL-92 Features: Basic schema manipulation

Feature	Source	Derby
CREATE / DROP TABLE	SQL92T	Yes
CREATE / DROP VIEW	SQL92T	Yes
GRANT / REVOKE	SQL92T	Yes
ALTER TABLE ADD COLUMN	SQL92T	Yes
ALTER TABLE DROP COLUMN	SQL92T	Yes

Joined table

Table 26. Support for SQL-92 Features: Joined table

Feature	Source	Derby
INNER JOIN	SQL92T	Yes
natural join	SQL92T	No
LEFT, RIGHT OUTER JOIN	SQL92T	Yes

Feature	Source	Derby
join condition	SQL92T	Yes
named columns join	SQL92T	Yes

Date and time data types

Table 27. Support for SQL-92 Features: Date and time data types

Feature	Source	Derby
simple DATE, TIME, TIMESTAMP, INTERVAL	SQL92T	Yes, not INTERVAL
datetime constants	SQL92T	Yes
datetime math	SQL92T	Yes, with Java methods
datetime comparisons	SQL92T	Yes
predicates: OVERLAPS	SQL92T	Yes, with Java methods

VARCHAR data type

Table 28. Support for SQL-92 Features: VARCHAR

Feature	Source	Derby
LENGTH	SQL92T	Yes
concatenation (||)	SQL92T	Yes

Transaction isolation

Table 29. Support for SQL-92 Features: Transaction isolation

Feature	Source	Derby
READ WRITE / READ ONLY	SQL92T	By using JDBC, database properties, and storage media
RU, RC, RR, SER	SQL92T	Yes

Multiple schemas per user

Table 30. Support for SQL-92 Features: Multiple schemas per user

Feature	Source	Derby
SCHEMATA view	SQL92T	*SYS.SYSSCHEMAS*

Privilege tables

Table 31. Support for SQL-92 Features: Privilege tables

Feature	Source	Derby
TABLE_PRIVILEGES	SQL92T	No
COLUMNS_PRIVILEGES	SQL92T	No
USAGE_PRIVILEGES	SQL92T	No

Table operations

Table 32. Support for SQL-92 Features: Table operations

Feature	Source	Derby
UNION relaxations	SQL92I	Yes
EXCEPT	SQL92I	Yes
INTERSECT	SQL92I	Yes
CORRESPONDING	SQL92I	No

Schema definition statement

Table 33. Support for SQL-92 Features: Schema definition statement

Feature	Source	Derby
CREATE SCHEMA	SQL92I	Partial support

User authorization

Table 34. Support for SQL-92 Features: User authorization

Feature	Source	Derby
SET SESSION AUTHORIZATION	SQL92I	Use SET SCHEMA
CURRENT_USER	SQL92I	Yes
SESSION_USER	SQL92I	Yes
SYSTEM_USER	SQL92I	No

Constraint tables

Table 35. Support for SQL-92 Features: Constraint tables

Feature	Source	Derby
TABLE CONSTRAINTS	SQL92I	*SYS.SYSCONSTRAINTS*
REFERENTIAL CONSTRAINTS	SQL92I	*SYS.SYSFOREIGNKEYS*
CHECK CONSTRAINTS	SQL92I	*SYS.SYSCHECKS*

Documentation schema

Table 36. Support for SQL-92 Features: Documentation schema

Feature	Source	Derby
SQL_FEATURES	SQL92I/FIPS 127-2	Use JDBC *DatabaseMetaData*
SQL_SIZING	SQL92I/FIPS 127-2	Use JDBC *DatabaseMetaData*

Full DATETIME

Table 37. Support for SQL-92 Features: Full DATETIME

Feature	Source	Derby
precision for TIME and TIMESTAMP	SQL92F	Yes

Full character functions

Table 38. Support for SQL-92 Features: Full character functions

Feature	Source	Derby
POSITION expression	SQL92F	Use Java methods or LOCATE
UPPER/LOWER functions	SQL92F	Yes

Miscellaneous features

Table 39. Support for SQL-92 Features: Miscellaneous

Feature	Source	Derby
Delimited identifiers	SQL92E	Yes
Correlated subqueries	SQL92E	Yes
Insert, Update, Delete statements	SQL92E	Yes
Joins	SQL92E	Yes
Where qualifications	SQL92E	Yes
Group by	SQL92E	Yes
Having	SQL92E	Yes
Aggregate functions	SQL92E	Yes
Order by	SQL92E	Yes
Select expressions	SQL92E	Yes
Select *	SQL92E	Yes
SQLCODE	SQL92E	No, deprecated in SQL-92
SQLSTATE	SQL92E	Yes
UNION, INTERSECT, and EXCEPT in views	SQL92T	Yes
Implicit numeric casting	SQL92T	Yes
Implicit character casting	SQL92T	Yes
Get diagnostics	SQL92T	Use JDBC *SQLExceptions*
Grouped operations	SQL92T	Yes
Qualified * in select list	SQL92T	Yes
Lowercase identifiers	SQL92T	Yes
nullable PRIMARY KEYs	SQL92T	No
Multiple module support	SQL92T	No (not required and not part of JDBC)

Feature	Source	Derby
Referential delete actions	SQL92T	CASCADE, SET NULL, RESTRICT, and NO ACTION
CAST functions	SQL92T	Yes
INSERT expressions	SQL92T	Yes
Explicit defaults	SQL92T	Yes
Keyword relaxations	SQL92T	Yes
Domain definition	SQL92I	No
CASE expression	SQL92I	Partial support
Compound character string constants	SQL92I	Use concatenation
LIKE enhancements	SQL92I	Yes
UNIQUE predicate	SQL92I	No
Usage tables	SQL92I	*SYS.SYSDEPENDS*
Intermediate information schema	SQL92I	Use JDBC *DatabaseMetaData* and Derby system tables
Subprogram support	SQL92I	Not relevant to JDBC, which is much richer
Intermediate SQL Flagging	SQL92I	No
Schema manipulation	SQL92I	Yes
Long identifiers	SQL92I	Yes
Full outer join	SQL92I	No
Time zone specification	SQL92I	No
Scrolled cursors	SQL92I	Partial support (scrollable insensitive result sets through JDBC 2.0)
Intermediate set function support	SQL92I	Partial support
Character set definition	SQL92I	Support for Java locales
Named character sets	SQL92I	Support for Java locales
Scalar subquery values	SQL92I	Yes
Expanded null predicate	SQL92I	Yes
Constraint management	SQL92I	Yes (ADD/DROP CONSTRAINT)
FOR BIT DATA types	SQL92F	Yes
Assertion constraints	SQL92F	No
Temporary tables	SQL92F	Partial support, with DECLARE GLOBAL TEMPORARY TABLE

Feature	Source	Derby
Full dynamic SQL	SQL92F	No
Full value expressions	SQL92F	Yes
Truth value tests	SQL92F	Yes
Derived tables in FROM	SQL92F	Yes
Trailing underscore	SQL92F	Yes
Indicator data types	SQL92F	Not relevant to JDBC
Referential name order	SQL92F	No
Full SQL Flagging	SQL92F	No
Row and table constructors	SQL92F	Yes
Catalog name qualifiers	SQL92F	No
Simple tables	SQL92F	No
Subqueries in CHECK	SQL92F	No, but can with Java methods
Union join	SQL92F	No
Collation and translation	SQL92F	Java locales supported
Referential update actions	SQL92F	RESTRICT and NO ACTION. Can do others with triggers.
ALTER domain	SQL92F	nNo
INSERT column privileges	SQL92F	No
Referential MATCH types	SQL92F	No
View CHECK enhancements	SQL92F	No, views cannot be updated
Session management	SQL92F	Use JDBC
Connection management	SQL92F	Use JDBC
Self-referencing operations	SQL92F	Yes
Insensitive cursors	SQL92F	Yes through JDBC 2.0
Full set function	SQL92F	Partial support
Catalog flagging	SQL92F	No
Local table references	SQL92F	No
Full cursor update	SQL92F	No

Derby system tables

Derby includes system tables.

You can query system tables, but you cannot alter them.

All of the above system tables reside in the *SYS* schema. Because this is not the default schema, qualify all queries accessing the system tables with the *SYS* schema name.

The recommended way to get more information about these tables is to use an instance of the Java interface *java.sql.DatabaseMetaData*.

SYSALIASES system table

Describes the procedures and functions in the database.

Column Name	Type	Length	Nullability	Contents
ALIASID	CHAR	36	false	unique identifier for the alias
ALIAS	VARCHAR	128	false	alias
SCHEMAID	CHAR	36	true	reserved for future use
JAVACLASSN	LONGVARCHAR	255	false	the Java class name
ALIASTYPE	CHAR	1	false	*'F'* (function)*'P'* (procedure)
NAMESPACE	CHAR	1	false	*'F'* (function)*'P'* (procedure)
SYSTEMALIAS	BOOLEAN	'	false	*true* (system supplied or built-in alias) *false* (alias created by a user)
ALIASINFO	org.apache.derby. catalog.AliasInfo: This class is not part of the public API	'	true	A Java interface that encapsulates the additional information that is specific to an alias
SPECIFICNAM	VARCHAR	128	false	system-generated identifier

SYSCHECKS system table

Describes the check constraints within the current database.

Column Name	Type	Length	Nullability	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for the constraint

Column Name	Type	Length	Nullability	Contents
CHECKDEFINITION	LONG VARCHAR	'	false	text of check constraint definition
REFERENCEDCOLUMNS	org.apache.derby.catalog.ReferencedColumns This class is not part of the public API.	'	false	description of the columns referenced by the check constraint

SYSCOLPERMS system table

The SYSCOLPERMS table stores the column permissions that have been granted but not revoked.

All of the permissions for one (GRANTEE, TABLEID, TYPE, GRANTOR) combination are specified in a single row in the SYSCOLPERMS table. The keys for the SYSCOLPERMS table are:

- Primary key (GRANTEE, TABLEID, TYPE, GRANTOR)
- Unique key (COLPERMSID)
- Foreign key (TABLEID references SYS.SYSTABLES)

Column Name	Type	Length	Nullability	Contents
COLPERMSID	CHAR	36	False	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the column level permissions.
GRANTEE	VARCHAR	128	False	The authorization ID of the user or role to which the privilege was granted.
GRANTOR	VARCHAR	128	False	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
TABLEID	CHAR	36	False	The unique identifier for the table on which the permissions have been granted.
TYPE	CHAR	1	False	If the privilege is non-grantable, the valid values are: 's' for SELECT 'u' for UPDATE 'r' for REFERENCES If the privilege is grantable, the valid values are: 'S' for SELECT 'U' for UPDATE 'R' for REFERENCES

Column Name	Type	Length	Nullability	Contents
COLUMNS	org.apa	'	False	A list of columns to which the privilege applies. This class is not part of the public API.

SYSCOLUMNS system table

Describes the columns within all tables in the current database:

Column Name	Type	Length	Nullable	Contents
REFERENCEID	CHAR	36	false	Identifier for table (join with *SYSTABLES.TABLEID*)
COLUMNNAME	CHAR	128	false	column or parameter name
COLUMNNUMBER	INT	4	false	the position of the column within the table
COLUMNDATATYPE	org.apache.derby.c TypeDescriptor This class is not part of the public API.	'	false	system type that describes precision, length, scale, nullability, type name, and storage type of data
COLUMNDEFAULT	*java.io.Serializable*	'	true	for tables, describes default value of the column. The *toString()* method on the object stored in the table returns the text of the default value as specified in the CREATE TABLE or ALTER TABLE statement.
COLUMNDEFAULTID	CHAR	36	true	unique identifier for the default value
AUTOINCREMENT COLUMNVALUE	BIGINT	'	true	what the next value for column will be, if the column is an identity column
AUTOINCREMENT COLUMNSTART	BIGINT	'	true	initial value of column (if specified), if it is an identity column
AUTOINCREMENT COLUMNINC	BIGINT	'	true	amount column value is automatically incremented (if specified), if the

Column Name	Type	Length	Nullable	Contents
				column is an identity column

SYSCONGLOMERATES system table

Describes the conglomerates within the current database. A conglomerate is a unit of storage and is either a table or an index.

Column Name	Type	Length	Nullable	Contents
SCHEMAID	CHAR	36	false	schema id for the conglomerate
TABLEID	CHAR	36	false	identifier for table (join with *SYSTABLES.TABLEID*)
CONGLOMERATENUMBE	BIGINT	8	false	conglomerate id for the conglomerate (heap or index)
CONGLOMERATENAME	VARCHAR	128	true	index name, if conglomerate is an index, otherwise the table ID
ISINDEX	BOOLEAN	1	false	whether or not conglomerate is an index
DESCRIPTOR	org.apache.derby. catalog.IndexDesc This class is not part of the public API.	'	true	system type describing the index
ISCONSTRAINT	BOOLEAN	1	true	whether or not conglomerate is a system-generated index enforcing a constraint
CONGLOMERATEID	CHAR	36	false	unique identifier for the conglomerate

SYSCONSTRAINTS system table

Describes the information common to all types of constraints within the current database (currently, this includes primary key, unique, foreign key, and check constraints).

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for constraint
TABLEID	CHAR	36	false	identifier for table (join with *SYSTABLES.TABLEID*)
CONSTRAINTNAME	VARCHAR	128	false	constraint name (internally generated if not specified by user)
TYPE	CHAR	1	false	*P* (primary key), *U* (unique), *C* (check), or *F* (foreign key)
SCHEMAID	CHAR	36	false	identifier for schema that the constraint belongs to (join with *SYSSCHEMAS.SCHEMAID*)
STATE	CHAR	1	false	*E* for enabled, *D* for disabled
REFERENCECOUNT	INTEGER	1	false	the count of the number of foreign key constraints that reference this constraint; this number can be greater than zero only for PRIMARY KEY and UNIQUE constraints

SYSDEPENDS system table

The SYSDEPENDS table stores the dependency relationships between persistent objects in the database.

Persistent objects can be dependents or providers. Dependents are objects that depend on other objects. Providers are objects that other objects depend on.

- Dependents are views, constraints, or triggers.
- Providers are tables, conglomerates, constraints, or privileges.

Column Name	Type	Length	Nullable	Contents
DEPENDENTID	CHAR	36	false	A unique identifier for the dependent.
DEPENDENTFINDER	org.apache.derby.ca DependableFinder: This class is not part of the public API.	1	false	A system type that describes the view, constraint, or trigger that is the dependent.
PROVIDERID	CHAR	36	false	A unique identifier for the provider.
PROVIDERFINDER	org.apache.derby.ca DependableFinder This class is not part of the public API.	1	false	A system type that describes the table, conglomerate, constraint, and privilege that is the provider

SYSFILES system table

Describes jar files stored in the database.

Column Name	Type	Length	Nullability	Contents
FILEID	CHAR	36	false	unique identifier for the jar file
SCHEMAID	CHAR	36	false	ID of the jar file's schema (join with *SYSSCHEMAS. SCHEMAID*)
FILENAME	VARCHAR	128	false	SQL name of the jar file
GENERATIONID	BIGINT	'	false	Generation number for the file. When jar files are replaced, their generation identifiers are changed.

SYSFOREIGNKEYS system table

Describes the information specific to foreign key constraints in the current database.

Derby generates a backing index for each foreign key constraint; the name of this index is the same as SYSFOREIGNKEYS.CONGLOMERATEID.

Column Name	Type	Length	Nullability	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for the foreign key constraint (join with *SYSCONSTRAINTS. CONSTRAINTID*)
CONGLOMERATEID	CHAR	36	false	unique identifier for index backing up the foreign key constraint (join with *SYSCONGLOMERATES. CONGLOMERATEID*)
KEYCONSTRAINTID	CHAR	36	false	unique identifier for the primary key or unique constraint referenced by this foreign key (*SYSKEYS.CONSTRAINTID* or *SYSCONSTRAINTS. CONSTRAINTID*)
DELETERULE	CHAR	1	false	*R* for NO ACTION (default), *S* for RESTRICT, *C* for CASCADE, *U* for SET NULL
UPDATERULE	CHAR	1	false	*R for NO ACTION(default), S* for restrict

SYSKEYS system table

Describes the specific information for primary key and unique constraints within the current database. Derby generates an index on the table to back up each such constraint. The index name is the same as *SYSKEYS.CONGLOMERATEID*.

Column Name	Type	Length	Nullable	Contents
CONSTRAINTID	CHAR	36	false	unique identifier for constraint
CONGLOMERATEID	CHAR	36	false	unique identifier for backing index

SYSROLES system table

The SYSROLES table stores the roles in the database.

A row in the SYSROLES table represents one of the following:

- A role definition (the result of a CREATE ROLE statement)
- A role grant

The keys for the SYSROLES table are:

- Primary key (GRANTEE, ROLEID, GRANTOR)
- Unique key (UUID)

Column Name	Type	Length	Nullability	Contents
UUID	CHAR	36	False	A unique identifier for this role.
ROLEID	VARCH	128	False	The role name, after conversion to case normal form.
GRANTEE	VARCH	128	False	If the row represents a role grant, this is the authorization identifier of a user or role to which this role is granted. If the row represents a role definition, this is the database owner's user name.
GRANTOR	VARCH	128	False	This is the authorization identifier of the user that granted this role. If the row represents a role definition, this is the authorization identifier _SYSTEM. If the row represents a role grant, this is the database owner's user name (since only the database owner can create and grant roles).
WITHADMINOPTION	CHAR	1	False	A role definition is modelled as a grant from _SYSTEM to the database owner, so if the row represents a role definition, the

Column Name	Type	Length	Nullability	Contents
				value is always 'Y'. This means that the creator (the database owner) is always allowed to grant the newly created role. Currently roles cannot be granted WITH ADMIN OPTION, so if the row represents a role grant, the value is always 'N'.
ISDEF	CHAR	1	False	If the row represents a role definition, this value is 'Y'. If the row represents a role grant, the value is 'N'.

SYSROUTINEPERMS system table

The SYSROUTINEPERMS table stores the permissions that have been granted to routines.

Each routine EXECUTE permission is specified in a row in the SYSROUTINEPERMS table. The keys for the SYSROUTINEPERMS table are:

- Primary key (GRANTEE, ALIASID, GRANTOR)
- Unique key (ROUTINEPERMSID)
- Foreign key (ALIASID references SYS.SYSALIASES)

The column information for the SYSTABLEPERMS table is listed in the following table:

Column Name	Type	Length	Nullability	Contents
ROUTINEPERMSID	CHAR	36	false	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the routine level permissions.
GRANTEE	VARCH	128	false	The authorization ID of the user or role to which the privilege is granted.
GRANTOR	VARCH	128	false	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
ALIASID	CHAR	36	false	The ID of the object of the required permission. If PERMTYPE='E' the ALIASID is a reference to the SYS.SYSALIASES table. Otherwise the ALIASID is a reference to the SYS.SYSTABLES table.
GRANTOPTION	CHAR	1	false	Specifies if the GRANTEE is the owner of the routine. Valid values are Y and N.

SYSSCHEMAS system table

Describes the schemas within the current database.

Column Name	Type	Length	Nullability	Contents
SCHEMAID	CHAR	36	false	unique identifier for the schema
SCHEMANAME	VARCHAR	128	false	schema name
AUTHORIZATIONID	VARCHAR	128	false	the authorization identifier of the owner of the schema

SYSSTATISTICS system table

Describes the schemas within the current database.

Column Name	Type	Length	Nullability	Contents
STATID	CHAR	36	false	unique identifier for the statistic
REFERENCEID	CHAR	36	false	the conglomerate for which the statistic was created (join with SYSCONGLOMERATES. CONGLOMERATEID)
TABLEID	CHAR	36	false	the table for which the information is collected
CREATIONTIMESTA	TIMESTAM	'	false	time when this statistic was created or updated
TYPE	CHAR	1	false	type of statistics
VALID	BOOLEAN	'	false	whether the statistic is still valid
COLCOUNT	INTEGER	'	false	number of columns in the statistic
STATISTICS	org.apache. derby.catalo Statistics: This class is not part of the public API.	'	true	statistics information

SYSSTATEMENTS system table

Contains one row per stored prepared statement.

Column Name	Type	Length	Nullability	Contents
STMTID	CHAR	36	false	unique identifier for the statement
STMTNAME	VARCHAR	128	false	name of the statement
SCHEMAID	CHAR	36	false	the schema in which the statement resides
TYPE	CHAR	1	false	always 'S'
VALID	BOOLEAN	'	false	TRUE if valid, FALSE if invalid
TEXT	LONG VARCHAR	'	false	text of the statement
LASTCOMPILED	TIMESTAMP	'	true	time that the statement was compiled
COMPILATION SCHEMAID	CHAR	36	false	id of the schema containing the statement
USINGTEXT	LONG VARCHAR	'	true	text of the USING clause of the CREATE STATEMENT and ALTER STATEMENT statements

SYSTABLEPERMS system table

The SYSTABLEPERMS table stores the table permissions that have been granted but not revoked.

All of the permissions for one (GRANTEE, TABLEID, GRANTOR) combination are specified in a single row in the SYSTABLEPERMS table. The keys for the SYSTABLEPERMS table are:

- Primary key (GRANTEE, TABLEID, GRANTOR)
- Unique key (TABLEPERMSID)
- Foreign key (TABLEID references SYS.SYSTABLES)

The column information for the SYSTABLEPERMS table is listed in the following table:

Column Name	Type	Length	Nullability	Contents
TABLEPERMSID	CHAR	36	False	Used by the dependency manager to track the dependency of a view, trigger, or constraint on the table level permissions.
GRANTEE	VARCH	128	False	The authorization ID of the user or role to which the privilege is granted.

Column Name	Type	Length	Nullability	Contents
GRANTOR	VARCH	128	False	The authorization ID of the user who granted the privilege. Privileges can be granted only by the object owner.
TABLEID	CHAR	36	False	The unique identifier for the table on which the permissions have been granted.
SELECTPRIV	CHAR	1	False	Specifies if the SELECT permission is granted. The valid values are: 'y' (non-grantable privilege) 'Y' (grantable privilege) 'N' (no privilege)
DELETEPRIV	CHAR	1	False	Specifies if the DELETE permission is granted. The valid values are: 'y' (non-grantable privilege) 'Y' (grantable privilege) 'N' (no privilege)
INSERTPRIV	CHAR	1	False	Specifies if the INSERT permission is granted. 'y' (non-grantable privilege) 'Y' (grantable privilege) 'N' (no privilege)
UPDATEPRIV	CHAR	1	False	Specifies if the UPDATE permission is granted. The valid values are: 'y' (non-grantable privilege) 'Y' (grantable privilege) 'N' (no privilege)
REFERENCEPRIV	CHAR	1	False	Specifies if the REFERENCE permission is granted. The valid values are: 'y' (non-grantable privilege) 'Y' (grantable privilege) 'N' (no privilege)
TRIGGERPRIV	CHAR	1	False	Specifies if the TRIGGER permission is granted. The valid values are: 'y' (non-grantable privilege) 'Y' (grantable privilege) 'N' (no privilege)

SYSTABLES system table

Describes the tables and views within the current database.

Column Name	Type	Length	Nullable	Contents
TABLEID	CHAR	36	false	unique identifier for table or view
TABLENAME	VARCHAR	128	false	table or view name
TABLETYPE	CHAR	1	false	'*S*' (system table), '*T*' (user table), '*A*' (synonym), or '*V*' (view)
SCHEMAID	CHAR	36	false	schema id for the table or view
LOCK GRANUL	CHAR	1	false	Indicates the lock granularity for the table ***'T'*** (table level locking) ***'R'*** (row level locking, the default)

SYSTRIGGERS system table

Describes the database's triggers.

Column Name	Type	Lengt	Nullabi	Contents
TRIGGERID	CHAR	36	false	unique identifier for the trigger
TRIGGERNAME	VARCHAR	128	false	name of the trigger
SCHEMAID	CHAR	36	false	id of the trigger's schema (join with *SYSSCHEMAS. SCHEMAID*)
CREATIONTIMESTAMI	TIMESTAMP	'	false	time the trigger was created
EVENT	CHAR	1	false	'U' for update, 'D' for delete, 'I' for insert
FIRINGTIME	CHAR	1	false	'B' for before 'A' for after
TYPE	CHAR	1	false	'R' for row, 'S' for statement
STATE	CHAR	1	false	'E' for enabled, 'D' for disabled
TABLEID	CHAR	36	false	id of the table on which the trigger is defined
WHENSTMTID	CHAR	36	true	used only if there is a WHEN clause (not yet supported)
ACTIONSTMTID	CHAR	36	true	id of the stored prepared statement for the triggered-SQL-statement (join with

Column Name	Type	Lengt	Nullabi	Contents
				SYSSTATEMENTS. STMTID)
REFERENCEDCOLUM	org.apache.derby.c ReferencedColum This class is not part of the public API.	'	true	descriptor of the columns referenced by UPDATE triggers
TRIGGERDEFINITION	LONG VARCHAR	'	true	text of the action SQL statement
REFERENCINGOLD	BOOLEAN	'	true	whether or not the OLDREFERENCINGNA if non-null, refers to the OLD row or table
REFERENCINGNEW	BOOLEAN	'	true	whether or not the NEWREFERENCINGNA if non-null, refers to the NEW row or table
OLDREFERENCINGNA	VARCHAR	128	true	pseudoname as set using the REFERENCING OLD AS clause
NEWREFERENCINGNA	VARCHAR	128	true	pseudoname as set using the REFERENCING NEW AS clause

Any SQL text that is part of a triggered-SQL-statement is compiled and stored in *SYSSTATEMENTS*. *ACTIONSTMTID* and *WHENSTMTID* are foreign keys that reference *SYSSTATEMENTS.STMTID*. The statements for a trigger are always in the same schema as the trigger.

SYSVIEWS system table

Describes the view definitions within the current database.

Column Name	Type	Length	Nullability	Contents
TABLEID	CHAR	36	false	unique identifier for the view (called TABLEID since it is joined with column of that name in SYSTABLES)
VIEWDEFINITION	LONG VARCHAR	'	false	text of view definition
CHECKOPTION	CHAR	1	false	'*N*' (check option not supported yet)
COMPILATION SCHEMAID	CHAR	36	false	id of the schema containing the view

Derby exception messages and SQL states

The JDBC driver returns *SQLExceptions* for all errors from Derby. If the exception originated in a user type but is not itself an *SQLException*, it is wrapped in an *SQLException*. Derby-specific *SQLExceptions* use *SQLState* class codes starting with *X*. Standard *SQLState* values are returned for exceptions where appropriate.

Unimplemented aspects of the JDBC driver return a *SQLException* with a *SQLState* starting with `0A`. If your application runs on JDK 1.6 or higher, then the exception class is *java.sql.SQLFeatureNotSupportedException*. These unimplemented parts are for features not supported by Derby.

Derby supplies values for the message and *SQLState* fields. In addition, Derby sometimes returns multiple *SQLExceptions* using the *nextException* chain. The first exception is always the most severe exception, with SQL-92 Standard exceptions preceding those that are specific to Derby.

For information on processing *SQLExceptions*, see the *Java DB Developer's Guide*.

SQL error messages and exceptions

The following tables list *SQLStates* for exceptions. Exceptions that begin with an *X* are specific to Derby.

Table 40. Class 01: Warning

SQLSTATE	Message Text
01001	An attempt to update or delete an already deleted row was made: No row was updated or deleted.
01003	Null values were eliminated from the argument of a column function.
01006	Privilege not revoked from user <*authorizationID*>.
01007	Role <*authorizationID*> not revoked from authentication id <*authorizationID*>.
01008	WITH ADMIN OPTION of role <*authorizationID*> not revoked from authentication id <*authorizationID*>.
01009	Generated column <*columnName*> dropped from table <*tableName*>.
0100E	XX Attempt to return too many result sets.
01500	The constraint <*constraintName*> on table <*tableName*> has been dropped.
01501	The view <*viewName*> has been dropped.
01502	The trigger <*triggerName*> on table <*tableName*> has been dropped.
01503	The column <*columnName*> on table <*tableName*> has been modified by adding a not null constraint.
01504	The new index is a duplicate of an existing index: <*indexName*>.
01505	The value <*valueName*> may be truncated.
01522	The newly defined synonym '<*synonymName*>' resolved to the object '<*objectName*>' which is currently undefined.
01J01	Database '<*databaseName*>' not created, connection made to existing database instead.

SQLSTATE	Message Text
01J02	Scroll sensitive cursors are not currently implemented.
01J04	The class '<*className*>' for column '<*columnName*>' does not implement java.io.Serializable or java.sql.SQLData. Instances must implement one of these interfaces to allow them to be stored.
01J05	Database upgrade succeeded. The upgraded database is now ready for use. Revalidating stored prepared statements failed. See next exception for details of failure.
01J06	ResultSet not updatable. Query does not qualify to generate an updatable ResultSet.
01J07	ResultSetHoldability restricted to ResultSet.CLOSE_CURSORS_AT_COMMIT for a global transaction.
01J08	Unable to open resultSet type <*resultSetType*>. ResultSet type <*resultSetType*> opened.
01J10	Scroll sensitive result sets are not supported by server; remapping to forward-only cursor
01J12	Unable to obtain message text from server. See the next exception. The stored procedure SYSIBM.SQLCAMESSAGE is not installed on the server. Please contact your database administrator.
01J13	Number of rows returned (<*number*>) is too large to fit in an integer; the value returned will be truncated.
01J14	SQL authorization is being used without first enabling authentication.

Table 41. Class 07: Dynamic SQL Error

SQLSTATE	Message Text
07000	At least one parameter to the current statement is uninitialized.
07004	Parameter <*parameterName*> is an <*procedureName*> procedure parameter and must be registered with CallableStatement.registerOutParameter before execution.
07009	No input parameters.

Table 42. Class 08: Connection Exception

SQLSTATE	Message Text
08000	Connection closed by unknown interrupt.
08001	A connection could not be established because the security token is larger than the maximum allowed by the network protocol.
08001	A connection could not be established because the user id has a length of zero or is larger than the maximum allowed by the network protocol.
08001	A connection could not be established because the password has a length of zero or is larger than the maximum allowed by the network protocol.
08001	Required Derby DataSource property <*propertyName*> not set.
08001	<*error*> : Error connecting to server <*serverName*> on port <*portNumber*> with message <*messageText*>.

SQLSTATE	Message Text
08001	SocketException: '<*error*>'
08001	Unable to open stream on socket: '<*error*>'.
08001	User id length (<*number*>) is outside the range of 1 to <*number*>.
08001	Password length (<*value*>) is outside the range of 1 to <*number*>.
08001	User id can not be null.
08001	Password can not be null.
08001	A connection could not be established because the database name '<*databaseName*>' is larger than the maximum length allowed by the network protocol.
08003	No current connection.
08003	getConnection() is not valid on a closed PooledConnection.
08003	Lob method called after connection was closed
08003	The underlying physical connection is stale or closed.
08004	Connection refused : <*connectionName*>
08004	Connection authentication failure occurred. Reason: <*reasonText*>.
08004	The connection was refused because the database <*databaseName*> was not found.
08004	Database connection refused.
08004	User '<*authorizationID*>' cannot shut down database '<*databaseName*>'. Only the database owner can perform this operation.
08004	User '<*authorizationID*>' cannot (re)encrypt database '<*databaseName*>'. Only the database owner can perform this operation.
08004	User '<*authorizationID*>' cannot hard upgrade database '<*databaseName*>'. Only the database owner can perform this operation.
08004	Connection refused to database '<*databaseName*>' because it is in replication slave mode.
08004	User '<*authorizationID*>' cannot issue a replication operation on database '<*databaseName*>'. Only the database owner can perform this operation.
08004	Missing permission for user '<*authorizationID*>' to shutdown system [<*exceptionMsg*>].
08004	Cannot check system permission to create database '<*databaseName*>' [<*exceptionMsg*>].
08004	Missing permission for user '<*authorizationID*>' to create database '<*databaseName*>' [<*exceptionMsg*>].
08006	A network protocol error was encountered and the connection has been terminated: <*error*>
08006	An error occurred during connect reset and the connection has been terminated. See chained exceptions for details.
08006	SocketException: '<*error*>'
08006	A communications error has been detected: <*error*>.

SQLSTATE	Message Text
08006	An error occurred during a deferred connect reset and the connection has been terminated. See chained exceptions for details.
08006	Insufficient data while reading from the network - expected a minimum of <*number*> bytes and received only <*number*> bytes. The connection has been terminated.
08006	Attempt to fully materialize lob data that is too large for the JVM. The connection has been terminated.
08006	Database '<*databaseName*>' shutdown.

Table 43. Class 0A: Feature not supported

SQLSTATE	Message Text
0A000	Feature not implemented: <*featureName*>.
0A000	The DRDA command <*commandName*> is not currently implemented. The connection has been terminated.
0A000	JDBC method is not yet implemented.
0A000	JDBC method <*methodName*> is not supported by the server. Please upgrade the server.
0A000	resultSetHoldability property <*propertyName*> not supported
0A000	cancel() not supported by the server.
0A000	Security mechanism '<*mechanismName*>' is not supported.
0A000	The data type '<*datatypeName*>' is not supported.

Table 44. Class 0P: Invalid role specification

SQLSTATE	Message Text
0P000	Invalid role specification, role does not exist: '<*roleName*>'.
0P000	Invalid role specification, role not granted to current user or PUBLIC: '<*roleName*>'.

Table 45. Class 21: Cardinality Violation

SQLSTATE	Message Text
21000	Scalar subquery is only allowed to return a single row.

Table 46. Class 22: Data Exception

SQLSTATE	Message Text
22001	A truncation error was encountered trying to shrink <*value*> '<*value*>' to length <*value*>.
22003	The resulting value is outside the range for the data type <*datatypeName*>.
22003	Year (<*value*>) exceeds the maximum '<*value*>'.
22003	Decimal may only be up to 31 digits.

SQLSTATE	Message Text
22003	Overflow occurred during numeric data type conversion of '<*datatypeName*>' to <*datatypeName*>.
22003	The length (<*number*>) exceeds the maximum length for the data type (<*datatypeName*>).
22005	Unable to convert a value of type '<*typeName*>' to type '<*typeName*>' : the encoding is not supported.
22005	The required character converter is not available.
22005	Unicode string cannot convert to Ebcdic string
22005	Unrecognized JDBC type. Type: <*typeName*>, columnCount: <*value*>, columnIndex: <*value*>.
22005	Invalid JDBC type for parameter <*parameterName*>.
22005	Unrecognized Java SQL type <*datatypeName*>.
22005	An attempt was made to get a data value of type '<*datatypeName*>' from a data value of type '<*datatypeName*>'.
22007	The string representation of a datetime value is out of range.
22007	The syntax of the string representation of a datetime value is incorrect.
22008	'<*argument*>' is an invalid argument to the <*functionName*> function.
2200L	Values assigned to XML columns must be well-formed DOCUMENT nodes.
2200M	Invalid XML DOCUMENT: <*parserError*>
2200V	Invalid context item for <*operatorName*> operator; context items must be well-formed DOCUMENT nodes.
2200W	XQuery serialization error: Attempted to serialize one or more top-level Attribute nodes.
22011	The second or third argument of the SUBSTR function is out of range.
22011	The range specified for the substring with offset <*offset*> and len <*len*> is out of range for the String: <*str*>.
22012	Attempt to divide by zero.
22013	Attempt to take the square root of a negative number, '<*value*>'.
22014	The start position for LOCATE is invalid; it must be a positive integer. The index to start the search from is '<*fromString*>'. The string to search for is '<*startIndex*>'. The string to search from is '<*searchString*>'.
22015	Invalid data conversion: requested conversion would result in a loss of precision of <*value*>
22015	The '<*functionName*>' function is not allowed on the following set of types. First operand is of type '<*typeName*>'. Second operand is of type '<*typeName*>'. Third operand (start position) is of type '<*typeName*>'.
22018	Invalid character string format for type <*typeName*>.
22019	Invalid escape sequence, '<*sequenceName*>'. The escape string must be exactly one character. It cannot be a null or more than one character.
22020	Invalid trim string, '<*string*>'. The trim string must be exactly one character or NULL. It cannot be more than one character.

SQLSTATE	Message Text
22025	Escape character must be followed by escape character, '_', or '%'. It cannot be followed by any other character or be at the end of the pattern.
22027	The built-in TRIM() function only supports a single trim character. The LTRIM() and RTRIM() built-in functions support multiple trim characters.
22028	The string exceeds the maximum length of <*number*>.
22501	An ESCAPE clause of NULL returns undefined results and is not allowed.
2201X	Invalid row count for OFFSET, must be >= 0.
2201W	Invalid row count for FIRST/NEXT, must be >= 1.

Table 47. Class 23: Constraint Violation

SQLSTATE	Message Text
23502	Column '<*columnName*>' cannot accept a NULL value.
23503	<*constraintName*> on table '<*tableName*>' caused a violation of foreign key constraint '<*value*>' for key <*keyName*>. The statement has been rolled back.
23505	The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint or unique index identified by '<*value*>' defined on '<*value*>'.
23513	The check constraint '<*tableName*>' was violated while performing an INSERT or UPDATE on table '<*constraintName*>'.

Table 48. Class 24: Invalid Cursor State

SQLSTATE	Message Text
24000	Invalid cursor state - no current row.
24501	The identified cursor is not open.

Table 49. Class 25: Invalid Transaction State

SQLSTATE	Message Text
25001	Cannot close a connection while a transaction is still active.
25001	Invalid transaction state: active SQL transaction.
25501	Unable to set the connection read-only property in an active transaction.
25502	An SQL data change is not permitted for a read-only connection, user or database.
25503	DDL is not permitted for a read-only connection, user or database.
25505	A read-only user or a user in a read-only database is not permitted to disable read-only mode on a connection.

Table 50. Class 28: Invalid Authorization Specification

SQLSTATE	Message Text
28502	The user name '<*authorizationID*>' is not valid.

Table 51. Class 2D: Invalid Transaction Termination

SQLSTATE	Message Text
2D521	setAutoCommit(true) invalid during global transaction.
2D521	COMMIT or ROLLBACK invalid for application execution environment.

Table 52. Class 38: External Function Exception

SQLSTATE	Message Text
38000	The exception '<*exception*>' was thrown while evaluating an expression.
38001	The external routine is not allowed to execute SQL statements.
38002	The routine attempted to modify data, but the routine was not defined as MODIFIES SQL DATA.
38004	The routine attempted to read data, but the routine was not defined as READS SQL DATA.

Table 53. Class 39: External Routine Invocation Exception

SQLSTATE	Message Text
39004	A NULL value cannot be passed to a method which takes a parameter of primitive type '<*type*>'.

Table 54. Class 3B: Invalid SAVEPOINT

SQLSTATE	Message Text
3B001	SAVEPOINT, <*savepointName*> does not exist or is not active in the current transaction.
3B002	The maximum number of savepoints has been reached.
3B501	A SAVEPOINT with the passed name already exists in the current transaction.
3B502	A RELEASE or ROLLBACK TO SAVEPOINT was specified, but the savepoint does not exist.

Table 55. Class 40: Transaction Rollback

SQLSTATE	Message Text
40001	A lock could not be obtained due to a deadlock, cycle of locks and waiters is: <*lockCycle*>. The selected victim is XID : <*transactionID*>.
40XC0	Dead statement. This may be caused by catching a transaction severity error inside this statement.
40XD0	Container has been closed.

SQLSTATE	Message Text
40XD1	Container was opened in read-only mode.
40XD2	Container *<containerName>* cannot be opened; it either has been dropped or does not exist.
40XL1	A lock could not be obtained within the time requested
40XL2	A lock could not be obtained within the time requested. The lockTable dump is: *<tableDump>*
40XT0	An internal error was identified by RawStore module.
40XT1	An exception was thrown during transaction commit.
40XT2	An exception was thrown during rollback of a SAVEPOINT.
40XT4	An attempt was made to close a transaction that was still active. The transaction has been aborted.
40XT5	Exception thrown during an internal transaction.
40XT6	Database is in quiescent state, cannot activate transaction. Please wait for a moment till it exits the quiescent state.
40XT7	Operation is not supported in an internal transaction.

Table 56. Class 42: Syntax Error or Access Rule Violation

SQLSTATE	Message Text
42000	Syntax error or access rule violation; see additional errors for details.
42500	User '*<authorizationID>*' does not have *<permissionType>* permission on table '*<schemaNamet>*'.'*<tableName>*'.
42501	User '*<authorizationID>*' does not have *<permissionType>* permission on table '*<schemaNamet>*'.'*<tableName>*' for grant.
42502	User '*<authorizationID>*' does not have *<permissionType>* permission on column '*<columnName>*' of table '*<schemaName>*'.'*<tableName>*'.
42503	User '*<authorizationID>*' does not have *<permissionType>* permission on column '*<columnName>*' of table '*<schemaName>*'.'*<tableName>*' for grant.
42504	User '*<authorizationID>*' does not have execute permission on *<objectName>* '*<schemaName>*'.'*<tableName>*'.
42505	User '*<authorizationID>*' does not have execute permission on *<objectName>* '*<schemaName>*'.'*<tableName>*' for grant.
42506	User '*<authorizationID>*' is not the owner of *<objectName>* '*<schemaName>*'.'*<tableName>*'.
42507	User '*<authorizationID>*' can not perform the operation in schema '*<schemaName>*'.
42508	User '*<authorizationID>*' can not create schema '*<schemaName>*'. Only database owner could issue this statement.
42509	Specified grant or revoke operation is not allowed on object '*<objectName>*'.
4250A	User '*<authorizationID>*' does not have *<permissionName>* permission on object '*<schemaName>*'.'*<objectName>*'.
4250B	Invalid database authorization property '*<value>*=*<value>*'.

SQLSTATE	Message Text
4250C	User(s) '<*authorizationID*>' must not be in both read-only and full-access authorization lists.
4250D	Repeated user(s) '<*listName*>' in access list '<*authorizationID*>';
4250E	Internal Error: invalid <*authorizationID*> id in statement permission list.
4251A	Statement <*value*> can only be issued by database owner.
4251B	PUBLIC is reserved and cannot be used as a user identifier or role name.
4251C	Role <*authorizationID*> cannot be granted to <*authorizationID*> because this would create a circularity.
42601	In an ALTER TABLE statement, the column '<*columnName*>' has been specified as NOT NULL and either the DEFAULT clause was not specified or was specified as DEFAULT NULL.
42601	ALTER TABLE statement cannot add an IDENTITY column to a table.
42605	The number of arguments for function '<*functionName*>' is incorrect.
42606	An invalid hexadecimal constant starting with '<*number*>' has been detected.
42610	All the arguments to the COALESCE/VALUE function cannot be parameters. The function needs at least one argument that is not a parameter.
42611	The length, precision, or scale attribute for column, or type mapping '<*value*>' is not valid.
42613	Multiple or conflicting keywords involving the '<*clause*>' clause are present.
42621	A check constraint or generated column that is defined with '<*value*>' is invalid.
42622	The name '<*name*>' is too long. The maximum length is '<*number*>'.
42734	Name '<*name*>' specified in context '<*context*>' is not unique.
42802	The number of values assigned is not the same as the number of specified or implied columns.
42803	An expression containing the column '<*columnName*>' appears in the SELECT list and is not part of a GROUP BY clause.
42815	The replacement value for '<*value*>' is invalid.
42815	The data type, length or value of arguments '<*value*>' and '<*value*>' is incompatible.
42818	Comparisons between '<*type*>' and '<*type*>' are not supported. Types must be comparable. String types must also have matching collation. If collation does not match, a possible solution is to cast operands to force them to the default collation (e.g. SELECT tablename FROM sys.systables WHERE CAST(tablename AS VARCHAR(128)) = 'T1')
42820	The floating point literal '<*string*>' contains more than 30 characters.
42821	Columns of type '<*type*>' cannot hold values of type '<*type*>'.
42824	An operand of LIKE is not a string, or the first operand is not a column.
42831	'<*columnName*>' cannot be a column of a primary key or unique key because it can contain null values.

SQLSTATE	Message Text
42831	'<*columnName*>' cannot be a column of a primary key because it can contain null values.
42834	SET NULL cannot be specified because FOREIGN KEY '<*key*>' cannot contain null values.
42837	ALTER TABLE '<*tableName*>' specified attributes for column '<*columnName*>' that are not compatible with the existing column.
42846	Cannot convert types '<*type*>' to '<*type*>'.
42877	A qualified column name '<*columnName*>' is not allowed in the ORDER BY clause.
42878	The ORDER BY clause of a SELECT UNION statement only supports unqualified column references and column position numbers. Other expressions are not currently supported.
42879	The ORDER BY clause may not contain column '<*columnName*>', since the query specifies DISTINCT and that column does not appear in the query result.
4287A	The ORDER BY clause may not specify an expression, since the query specifies DISTINCT.
42884	No authorized routine named '<*routineName*>' of type '<*type*>' having compatible arguments was found.
42886	'<*value*>' parameter '<*value*>' requires a parameter marker '?'.
42894	DEFAULT value or IDENTITY attribute value is not valid for column '<*columnName*>'.
428C1	Only one identity column is allowed in a table.
428EK	The qualifier for a declared global temporary table name must be SESSION.
42903	Invalid use of an aggregate function.
42908	The CREATE VIEW statement does not include a column list.
42909	The CREATE TABLE statement does not include a column list.
42915	Foreign Key '<*key*>' is invalid because '<*value*>'.
42916	Synonym '<*synonym2*>' cannot be created for '<*synonym1*>' as it would result in a circular synonym chain.
42939	An object cannot be created with the schema name '<*schemaNamet*>'.
4293A	A role cannot be created with the name '<*authorizationID*>', the SYS prefix is reserved.
42962	Long column type column or parameter '<*columnName*>' not permitted in declared global temporary tables or procedure definitions.
42972	An ON clause associated with a JOIN operator is not valid.
42995	The requested function does not apply to global temporary tables.
42X01	Syntax error: <*error*>.
42X02	<*value*>.
42X03	Column name '<*columnName*>' is in more than one table in the FROM list.
42X04	

SQLSTATE	Message Text
	Column '<*columnName*>' is either not in any table in the FROM list or appears within a join specification and is outside the scope of the join specification or appears in a HAVING clause and is not in the GROUP BY list. If this is a CREATE or ALTER TABLE statement then '<*columnName*>' is not a column in the target table.
42X05	Table/View '<*objectName*>' does not exist.
42X06	Too many result columns specified for table '<*tableName*>'.
42X07	Null is only allowed in a VALUES clause within an INSERT statement.
42X08	The constructor for class '<*className*>' cannot be used as an external virtual table because the class does not implement '<*constructorName*>'.
42X09	The table or alias name '<*tableName*>' is used more than once in the FROM list.
42X10	'<*tableName*>' is not an exposed table name in the scope in which it appears.
42X12	Column name '<*columnName*>' appears more than once in the CREATE TABLE statement.
42X13	Column name '<*columnName*>' appears more than once times in the column list of an INSERT statement.
42X14	'<*columnName*>' is not a column in table or VTI '<*value*>'.
42X15	Column name '<*columnName*>' appears in a statement without a FROM list.
42X16	Column name '<*columnName*>' appears multiple times in the SET clause of an UPDATE statement.
42X17	In the Properties list of a FROM clause, the value '<*value*>' is not valid as a joinOrder specification. Only the values FIXED and UNFIXED are valid.
42X19	The WHERE or HAVING clause or CHECK CONSTRAINT definition is a '<*value*>' expression. It must be a BOOLEAN expression.
42X20	Syntax error; integer literal expected.
42X23	Cursor <*cursorName*> is not updatable.
42X24	Column <*columnName*> is referenced in the HAVING clause but is not in the GROUP BY list.
42X25	The '<*functionName*>' function is not allowed on the '<*1*>' type.
42X26	The class '<*className*>' for column '<*columnName*>' does not exist or is inaccessible. This can happen if the class is not public.
42X28	Delete table '<*tableName*>' is not target of cursor '<*cursorName*>'.
42X29	Update table '<*tableName*>' is not the target of cursor '<*cursorName*>'.
42X30	Cursor '<*cursorName*>' not found. Verify that autocommit is OFF.
42X31	Column '<*columnName*>' is not in the FOR UPDATE list of cursor '<*cursorName*>'.
42X32	The number of columns in the derived column list must match the number of columns in table '<*tableName*>'.
42X33	The derived column list contains a duplicate column name '<*columnName*>'.

SQLSTATE	Message Text
42X34	There is a ? parameter in the select list. This is not allowed.
42X35	It is not allowed for both operands of '<*value*>' to be ? parameters.
42X36	The '<*operator*>' operator is not allowed to take a ? parameter as an operand.
42X37	The unary '<*operator*>' operator is not allowed on the '<*type*>' type.
42X38	'SELECT *' only allowed in EXISTS and NOT EXISTS subqueries.
42X39	Subquery is only allowed to return a single column.
42X40	A NOT statement has an operand that is not boolean . The operand of NOT must evaluate to TRUE, FALSE, or UNKNOWN.
42X41	In the Properties clause of a FROM list, the property '<*propertyName*>' is not valid (the property was being set to '<*value*>').
42X42	Correlation name not allowed for column '<*columnName*>' because it is part of the FOR UPDATE list.
42X43	The ResultSetMetaData returned for the class/object '<*className*>' was null. In order to use this class as an external virtual table, the ResultSetMetaData cannot be null.
42X44	Invalid length '<*number*>' in column specification.
42X45	<*type*> is an invalid type for argument number <*value*> of <*value*>.
42X46	There are multiple functions named '<*functionName*>'. Use the full signature or the specific name.
42X47	There are multiple procedures named '<*procedureName*>'. Use the full signature or the specific name.
42X48	Value '<*value*>' is not a valid precision for <*value*>.
42X49	Value '<*value*>' is not a valid integer literal.
42X50	No method was found that matched the method call <*methodName*>.<*value*>(<*value*>), tried all combinations of object and primitive types and any possible type conversion for any parameters the method call may have. The method might exist but it is not public and/or static, or the parameter types are not method invocation convertible.
42X51	The class '<*className*>' does not exist or is inaccessible. This can happen if the class is not public.
42X52	Calling method ('<*methodName*>') using a receiver of the Java primitive type '<*type*>' is not allowed.
42X53	The LIKE predicate can only have 'CHAR' or 'VARCHAR' operands. Type '<*type*>' is not permitted.
42X54	The Java method '<*methodName*>' has a ? as a receiver. This is not allowed.
42X55	Table name '<*value*>' should be the same as '<*tableName*>'.
42X56	The number of columns in the view column list does not match the number of columns in the underlying query expression in the view definition for '<*value*>'.

SQLSTATE	Message Text
42X57	The getColumnCount() for external virtual table '<*tableName*>' returned an invalid value '<*value*>'. Valid values are greater than or equal to 1.
42X58	The number of columns on the left and right sides of the <*tableName*> must be the same.
42X59	The number of columns in each VALUES constructor must be the same.
42X60	Invalid value '<*value*>' for insertMode property specified for table '<*tableName*>'.
42X61	Types '<*type*>' and '<*type*>' are not <*value*> compatible.
42X62	'<*value*>' is not allowed in the '<*schemaNamet*>' schema.
42X63	The USING clause did not return any results. No parameters can be set.
42X64	In the Properties list, the invalid value '<*value*>' was specified for the useStatistics property. The only valid values are TRUE or FALSE.
42X65	Index '<*index*>' does not exist.
42X66	Column name '<*columnName*>' appears more than once in the CREATE INDEX statement.
42X68	No field '<*fieldName*>' was found belonging to class '<*className*>'. It may be that the field exists, but it is not public, or that the class does not exist or is not public.
42X69	It is not allowed to reference a field ('<*fieldName*>') using a referencing expression of the Java primitive type '<*type*>'.
42X70	The number of columns in the table column list does not match the number of columns in the underlying query expression in the table definition for '<*value*>'.
42X71	Invalid data type '<*datatypeName*>' for column '<*columnName*>'.
42X72	No static field '<*fieldName*>' was found belonging to class '<*className*>'. The field might exist, but it is not public and/or static, or the class does not exist or the class is not public.
42X73	Method resolution for signature <*value*>.<*value*>(<*value*>) was ambiguous. (No single maximally specific method.)
42X74	Invalid CALL statement syntax.
42X75	No constructor was found with the signature <*value*>(<*value*>). It may be that the parameter types are not method invocation convertible.
42X76	At least one column, '<*columnName*>', in the primary key being added is nullable. All columns in a primary key must be non-nullable.
42X77	Column position '<*columnPosition*>' is out of range for the query expression.
42X78	Column '<*columnName*>' is not in the result of the query expression.
42X79	Column name '<*columnName*>' appears more than once in the result of the query expression.
42X80	VALUES clause must contain at least one element. Empty elements are not allowed.
42X82	The USING clause returned more than one row. Only single-row ResultSets are permissible.

SQLSTATE	Message Text
42X83	The constraints on column '<*columnName*>' require that it be both nullable and not nullable.
42X84	Index '<*index*>' was created to enforce constraint '<*constraintName*>'. It can only be dropped by dropping the constraint.
42X85	Constraint '<*constraintName*>'is required to be in the same schema as table '<*tableName*>'.
42X86	ALTER TABLE failed. There is no constraint '<*constraintName*>' on table '<*tableName*>'.
42X87	At least one result expression (THEN or ELSE) of the '<*expression*>' expression must not be a '?'.
42X88	A conditional has a non-Boolean operand. The operand of a conditional must evaluate to TRUE, FALSE, or UNKNOWN.
42X89	Types '<*type*>' and '<*type*>' are not type compatible. Neither type is assignable to the other type.
42X90	More than one primary key constraint specified for table '<*tableName*>'.
42X91	Constraint name '<*constraintName*>' appears more than once in the CREATE TABLE statement.
42X92	Column name '<*columnName*>' appears more than once in a constraint's column list.
42X93	Table '<*tableName*>' contains a constraint definition with column '<*columnName*>' which is not in the table.
42X94	<*value*> '<*value*>' does not exist.
42X96	The database class path contains an unknown jar file '<*fileName*>'.
42X98	Parameters are not allowed in a VIEW definition.
42X99	Parameters are not allowed in a TABLE definition.
42XA0	The generation clause for column '<*columnName*>' has data type '<*datatypeName*>', which cannot be assigned to the column's declared data type.
42XA1	The generation clause for column '<*columnName*>' contains an aggregate. This is not allowed.
42XA2	'<*value*>' cannot appear in a GENERATION CLAUSE because it may return unreliable results.
42XA3	You may not override the value of generated column '<*columnName*>'.
42XA4	The generation clause for column '<*columnName*>' references other generated columns. This is not allowed.
42XA5	Routine '<*routineName*>' may issue SQL and therefore cannot appear in a GENERATION CLAUSE.
42XA6	'<*columnName*>' is a generated column. It cannot be part of a foreign key whose referential action for DELETE is SET NULL or SET DEFAULT, or whose referential action for UPDATE is CASCADE.
42XA7	'<*columnName*>' is a generated column. You cannot change its default value.

SQLSTATE	Message Text
42XA8	You cannot rename '<*columnName*>' because it is referenced by the generation clause of column '<*columnName*>'.
42XA9	Column '<*columnName*>' needs an explicit datatype. The datatype can be omitted only for columns with generation clauses.
42XAA	The NEW value of generated column '<*columnName*>' is mentioned in the BEFORE action of a trigger. This is not allowed.
42XAB	NOT NULL is allowed only if you explicitly declare a datatype.
42Y00	Class '<*className*>' does not implement org.apache.derby.iapi.db.AggregateDefinition and thus cannot be used as an aggregate expression.
42Y01	Constraint '<*constraintName*>' is invalid.
42Y03	'<*statement*>' is not recognized as a function or procedure.
42Y03	'<*statement*>' is not recognized as a procedure.
42Y03	'<*statement*>' is not recognized as a function.
42Y04	Cannot create a procedure or function with EXTERNAL NAME '<*name*>' because it is not a list separated by periods. The expected format is <full java path>.<method name>.
42Y05	There is no Foreign Key named '<*key*>'.
42Y07	Schema '<*schemaNamet*>' does not exist
42Y08	Foreign key constraints are not allowed on system tables.
42Y09	Void methods are only allowed within a CALL statement.
42Y10	A table constructor that is not in an INSERT statement has all ? parameters in one of its columns. For each column, at least one of the rows must have a non-parameter.
42Y11	A join specification is required with the '<*clauseName*>' clause.
42Y12	The ON clause of a JOIN is a '<*expressionType*>' expression. It must be a BOOLEAN expression.
42Y13	Column name '<*columnName*>' appears more than once in the CREATE VIEW statement.
42Y16	No public static method '<*methodName*>' was found in class '<*className*>'. The method might exist, but it is not public, or it is not static.
42Y22	Aggregate <*aggregateType*> cannot operate on type <*type*>.
42Y23	Incorrect JDBC type info returned for column <*colunmName*>.
42Y24	View '<*viewName*>' is not updatable. (Views are currently not updatable.)
42Y25	'<*tableName*>' is a system table. Users are not allowed to modify the contents of this table.
42Y26	Aggregates are not allowed in the GROUP BY list.
42Y27	Parameters are not allowed in the trigger action.
42Y29	The SELECT list of a non-grouped query contains at least one invalid expression. When the SELECT list contains at least one aggregate then all entries must be valid aggregate expressions.

SQLSTATE	Message Text
42Y30	The SELECT list of a grouped query contains at least one invalid expression. If a SELECT list has a GROUP BY, the list may only contain valid grouping expressions and valid aggregate expressions.
42Y32	Aggregator class '<*className*>' for aggregate '<*aggregateName*>' on type <*type*> does not implement org.apache.derby.iapi.sql.execute.ExecAggregator.
42Y33	Aggregate <*aggregateName*> contains one or more aggregates.
42Y34	Column name '<*columnName*>' matches more than one result column in table '<*tableName*>'.
42Y35	Column reference '<*reference*>' is invalid. When the SELECT list contains at least one aggregate then all entries must be valid aggregate expressions.
42Y36	Column reference '<*reference*>' is invalid, or is part of an invalid expression. For a SELECT list with a GROUP BY, the columns and expressions being selected may only contain valid grouping expressions and valid aggregate expressions.
42Y37	'<*value*>' is a Java primitive and cannot be used with this operator.
42Y38	insertMode = replace is not permitted on an insert where the target table, '<*tableName*>', is referenced in the SELECT.
42Y39	'<*value*>' may not appear in a CHECK CONSTRAINT definition because it may return non-deterministic results.
42Y40	'<*value*>' appears multiple times in the UPDATE OF column list for trigger '<*triggerName*>'.
42Y41	'<*value*>' cannot be directly invoked via EXECUTE STATEMENT because it is part of a trigger.
42Y42	Scale '<*value*>' is not a valid scale for a <*scaleValue*>.
42Y43	Scale '<*scaleValue*>' is not a valid scale with precision of '<*precision*>'.
42Y44	Invalid key '<*key*>' specified in the Properties list of a FROM list. The case-sensitive keys that are currently supported are '<*key*>'.
42Y45	VTI '<*value*>' cannot be bound because it is a special trigger VTI and this statement is not part of a trigger action or WHEN clause.
42Y46	Invalid Properties list in FROM list. There is no index '<*index*>' on table '<*tableName*>'.
42Y48	Invalid Properties list in FROM list. Either there is no named constraint '<*constraintName*>' on table '<*tableName*>' or the constraint does not have a backing index.
42Y49	Multiple values specified for property key '<*key*>'.
42Y50	Properties list for table '<*tableName*>' may contain values for index or for constraint but not both.
42Y55	'<*value*>' cannot be performed on '<*value*>' because it does not exist.
42Y56	Invalid join strategy '<*strategyValue*>' specified in Properties list on table '<*tableName*>'. The currently supported values for a join strategy are: 'hash' and 'nestedloop'.

SQLSTATE	Message Text
42Y58	NumberFormatException occurred when converting value '*<value>*' for optimizer override '*<value>*'.
42Y59	Invalid value, '*<value>*', specified for hashInitialCapacity override. Value must be greater than 0.
42Y60	Invalid value, '*<value>*', specified for hashLoadFactor override. Value must be greater than 0.0 and less than or equal to 1.0.
42Y61	Invalid value, '*<value>*', specified for hashMaxCapacity override. Value must be greater than 0.
42Y62	'*<statement>*' is not allowed on '*<viewName>*' because it is a view.
42Y63	Hash join requires an optimizable equijoin predicate on a column in the selected index or heap. An optimizable equijoin predicate does not exist on any column in table or index '*<index>*'. Use the 'index' optimizer override to specify such an index or the heap on table '*<tableName>*'.
42Y64	bulkFetch value of '*<value>*' is invalid. The minimum value for bulkFetch is 1.
42Y65	bulkFetch is not permitted on '*<joinType>*' joins.
42Y66	bulkFetch is not permitted on updatable cursors.
42Y67	Schema '*<schemaNamet>*' cannot be dropped.
42Y69	No valid execution plan was found for this statement. This may have one of two causes: either you specified a hash join strategy when hash join is not allowed (no optimizable equijoin) or you are attempting to join two external virtual tables, each of which references the other, and so the statement cannot be evaluated.
42Y70	The user specified an illegal join order. This could be caused by a join column from an inner table being passed as a parameter to an external virtual table.
42Y71	System function or procedure '*<procedureName>*' cannot be dropped.
42Y82	System generated stored prepared statement '*<statement>*' that cannot be dropped using DROP STATEMENT. It is part of a trigger.
42Y83	An untyped null is not permitted as an argument to aggregate *<aggregateName>*. Please cast the null to a suitable type.
42Y84	'*<value>*' may not appear in a DEFAULT definition.
42Y85	The DEFAULT keyword is only allowed in a VALUES clause when the VALUES clause appears within an INSERT statement.
42Y90	FOR UPDATE is not permitted in this type of statement.
42Y91	The USING clause is not permitted in an EXECUTE STATEMENT for a trigger action.
42Y92	*<triggerName>* triggers may only reference *<value>* transition variables/tables.
42Y93	Illegal REFERENCING clause: only one name is permitted for each type of transition variable/table.
42Y94	An AND or OR has a non-boolean operand. The operands of AND and OR must evaluate to TRUE, FALSE, or UNKNOWN.
42Y95	

SQLSTATE	Message Text
	The '<*operatorName*>' operator with a left operand type of '<*operandType*>' and a right operand type of '<*operandType*>' is not supported.
42Y97	Invalid escape character at line '<*lineNumber*>', column '<*columnName*>'.
42Z02	Multiple DISTINCT aggregates are not supported at this time.
42Z07	Aggregates are not permitted in the ON clause.
42Z08	Bulk insert replace is not permitted on '<*value*>' because it has an enabled trigger (<*value*>).
42Z15	Invalid type specified for column '<*columnName*>'. The type of a column may not be changed.
42Z16	Only columns of type VARCHAR may have their length altered.
42Z17	Invalid length specified for column '<*columnName*>'. Length must be greater than the current column length.
42Z18	Column '<*columnName*>' is part of a foreign key constraint '<*constraintName*>'. To alter the length of this column, you should drop the constraint first, perform the ALTER TABLE, and then recreate the constraint.
42Z19	Column '<*columnName*>' is being referenced by at least one foreign key constraint '<*constraintName*>'. To alter the length of this column, you should drop referencing constraints, perform the ALTER TABLE and then recreate the constraints.
42Z20	Column '<*columnName*>' cannot be made nullable. It is part of a primary key or unique constraint, which cannot have any nullable columns.
42Z20	Column '<*columnName*>' cannot be made nullable. It is part of a primary key, which cannot have any nullable columns.
42Z21	Invalid increment specified for identity for column '<*columnName*>'. Increment cannot be zero.
42Z22	Invalid type specified for identity column '<*columnName*>'. The only valid types for identity columns are BIGINT, INT and SMALLINT.
42Z23	Attempt to modify an identity column '<*columnName*>'.
42Z24	Overflow occurred in identity value for column '<*tableName*>' in table '<*columnName*>'.
42Z25	INTERNAL ERROR identity counter. Update was called without arguments with current value \= NULL.
42Z26	A column, '<*columnName*>', with an identity default cannot be made nullable.
42Z27	A nullable column, '<*columnName*>', cannot be modified to have identity default.
42Z50	INTERNAL ERROR: Unable to generate code for <*value*>.
42Z53	INTERNAL ERROR: Type of activation to generate for node choice <*value*> is unknown.
42Z60	<*value*> not allowed unless database property <*propertyName*> has value '<*value*>'.
42Z70	Binding directly to an XML value is not allowed; try using XMLPARSE.

SQLSTATE	Message Text
42Z71	XML values are not allowed in top-level result sets; try using XMLSERIALIZE.
42Z72	Missing SQL/XML keyword(s) '<*keywords*>' at line <*lineNumber*>, column <*columnNumber*>.
42Z73	Invalid target type for XMLSERIALIZE: '<*typeName*>'.
42Z74	XML feature not supported: '<*featureName*>'.
42Z75	XML query expression must be a string literal.
42Z76	Multiple XML context items are not allowed.
42Z77	Context item must have type 'XML'; '<*value*>' is not allowed.
42Z79	Unable to determine the parameter type for XMLPARSE; try using a CAST.
42Z90	Class '<*className*>' does not return an updatable ResultSet.
42Z91	subquery
42Z92	repeatable read
42Z93	Constraints '<*constraintName*>' and '<*constraintName*>' have the same set of columns, which is not allowed.
42Z97	Renaming column '<*columnName*>' will cause check constraint '<*constraintName*>' to break.
42Z99	String or Hex literal cannot exceed 64K.
42Z9A	read uncommitted
42Z9B	The external virtual table interface does not support BLOB or CLOB columns. '<*value*>' column '<*value*>'.
42Z9D	Procedures that modify SQL data are not allowed in BEFORE triggers.
42Z9D	'<*statement*>' statements are not allowed in '<*triggerName*>' triggers.
42Z9E	Constraint '<*constraintName*>' is not a <*value*> constraint.
42Z9F	Too many indexes (<*index*>) on the table <*tableName*>. The limit is <*number*>.
42ZA0	Statement too complex. Try rewriting the query to remove complexity. Eliminating many duplicate expressions or breaking up the query and storing interim results in a temporary table can often help resolve this error.
42ZA1	Invalid SQL in Batch: '<*batch*>'.
42ZA2	Operand of LIKE predicate with type <*type*> and collation <*value*> is not compatable with LIKE pattern operand with type <*type*> and collation <*value*>.
42ZA3	The table will have collation type <*type*> which is different than the collation of the schema <*type*> hence this operation is not supported .
42ZB1	Parameter style DERBY_JDBC_RESULT_SET is only allowed for table functions.
42ZB2	Table functions can only have parameter style DERBY_JDBC_RESULT_SET.
42ZB3	XML is not allowed as the datatype of a column returned by a table function.

SQLSTATE	Message Text
42ZB4	'<*schemaName*>'.<*functionName*>' does not identify a table function.
42ZB5	Class '<*className*>' implements VTICosting but does not provide a public, no-arg constructor.

Table 57. Class 57: DRDA Network Protocol: Execution Failure

SQLSTATE	Message Text
57017	There is no available conversion for the source code page, <*codePage*>, to the target code page, <*codePage*>. The connection has been terminated.

Table 58. Class 58: DRDA Network Protocol: Protocol Error

SQLSTATE	Message Text
58009	Network protocol exception: only one of the VCM, VCS length can be greater than 0. The connection has been terminated.
58009	The connection was terminated because the encoding is not supported.
58009	Network protocol exception: actual code point, <*value*>, does not match expected code point, <*value*>. The connection has been terminated.
58009	Network protocol exception: DDM collection contains less than 4 bytes of data. The connection has been terminated.
58009	Network protocol exception: collection stack not empty at end of same id chain parse. The connection has been terminated.
58009	Network protocol exception: DSS length not 0 at end of same id chain parse. The connection has been terminated.
58009	Network protocol exception: DSS chained with same id at end of same id chain parse. The connection has been terminated.
58009	Network protocol exception: end of stream prematurely reached while reading InputStream, parameter #<*value*>. The connection has been terminated.
58009	Network protocol exception: invalid FDOCA LID. The connection has been terminated.
58009	Network protocol exception: SECTKN was not returned. The connection has been terminated.
58009	Network protocol exception: only one of NVCM, NVCS can be non-null. The connection has been terminated.
58009	Network protocol exception: SCLDTA length, <*length*>, is invalid for RDBNAM. The connection has been terminated.
58009	Network protocol exception: SCLDTA length, <*length*>, is invalid for RDBCOLID. The connection has been terminated.
58009	Network protocol exception: SCLDTA length, <*length*>, is invalid for PKGID. The connection has been terminated.
58009	Network protocol exception: PKGNAMCSN length, <*length*>, is invalid at SQLAM <*value*>. The connection has been terminated.
58010	

SQLSTATE	Message Text
	A network protocol error was encountered. A connection could not be established because the manager <*value*> at level <*value*> is not supported by the server.
58014	The DDM command 0x<*value*> is not supported. The connection has been terminated.
58015	The DDM object 0x<*value*> is not supported. The connection has been terminated.
58016	The DDM parameter 0x<*value*> is not supported. The connection has been terminated.
58017	The DDM parameter value 0x<*value*> is not supported. An input host variable may not be within the range the server supports. The connection has been terminated.

Table 59. Class X0: Execution exceptions

SQLSTATE	Message Text
X0A00	The select list mentions column '<*columnName*>' twice. This is not allowed in queries with GROUP BY or HAVING clauses. Try aliasing one of the conflicting columns to a unique name.
X0X02	Table '<*tableName*>' cannot be locked in '<*mode*>' mode.
X0X03	Invalid transaction state - held cursor requires same isolation level
X0X05	Table/View '<*tableName*>' does not exist.
X0X07	Cannot remove jar file '<*fileName*>' because it is on your derby.database.classpath '<*fileName*>'.
X0X0D	Invalid column array length '<*columnArrayLength*>'. To return generated keys, column array must be of length 1 and contain only the identity column.
X0X0E	Table '<*tableName*>' does not have an auto-generated column at column position '<*columnPosition*>'.
X0X0F	Table '<*tableName*>' does not have an auto-generated column named '<*columnName*>'.
X0X10	The USING clause returned more than one row; only single-row ResultSets are permissible.
X0X11	The USING clause did not return any results so no parameters can be set.
X0X13	Jar file '<*fileName*>' does not exist in schema '<*schemaNamet*>'.
X0X57	An attempt was made to put a Java value of type '<*type*>' into a SQL value, but there is no corresponding SQL type. The Java value is probably the result of a method call or field access.
X0X60	A cursor with name '<*cursorName*>' already exists.
X0X61	The values for column '<*location*>' in index '<*columnName*>' and table '<*indexName*>.<*schemaNamet*>' do not match for row location <*tableName*>. The value in the index is '<*value*>', while the value in the base table is '<*value*>'. The full index key, including the row location, is '<*indexKey*>'. The suggested corrective action is to recreate the index.
X0X62	

SQLSTATE	Message Text
	Inconsistency found between table '<*tableName*>' and index '<*indexName*>'. Error when trying to retrieve row location '<*rowLocation*>' from the table. The full index key, including the row location, is '<*indexKey*>'. The suggested corrective action is to recreate the index.
X0X63	Got IOException '<*value*>'.
X0X67	Columns of type '<*type*>' may not be used in CREATE INDEX, ORDER BY, GROUP BY, UNION, INTERSECT, EXCEPT or DISTINCT statements because comparisons are not supported for that type.
X0X81	<*value*> '<*value*>' does not exist.
X0X85	Index '<*indexName*>' was not created because '<*indexType*>' is not a valid index type.
X0X86	0 is an invalid parameter value for ResultSet.absolute(int row).
X0X87	ResultSet.relative(int row) cannot be called when the cursor is not positioned on a row.
X0X95	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because there is an open ResultSet dependent on that object.
X0X99	Index '<*indexName*>' does not exist.
X0Y16	'<*value*>' is not a view. If it is a table, then use DROP TABLE instead.
X0Y23	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because VIEW '<*viewName*>' is dependent on that object.
X0Y24	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because STATEMENT '<*statement*>' is dependent on that object.
X0Y25	Operation '<*operationName*>' cannot be performed on object '<*objectName*>' because <*value*> '<*value*>' is dependent on that object.
X0Y26	Index '<*indexName*>' is required to be in the same schema as table '<*tableName*>'.
X0Y28	Index '<*indexName*>' cannot be created on system table '<*tableName*>'. Users cannot create indexes on system tables.
X0Y32	<*value*> '<*value*>' already exists in <*value*> '<*value*>'.
X0Y38	Cannot create index '<*indexName*>' because table '<*tableName*>' does not exist.
X0Y41	Constraint '<*constraintName*>' is invalid because the referenced table <*tableName*> has no primary key. Either add a primary key to <*tableName*> or explicitly specify the columns of a unique constraint that this foreign key references.
X0Y42	Constraint '<*constraintName*>' is invalid: the types of the foreign key columns do not match the types of the referenced columns.
X0Y43	Constraint '<*constraintName*>' is invalid: the number of columns in <*constraintName*> (<*value*>) does not match the number of columns in the referenced key (<*value*>).
X0Y44	Constraint '<*constraintName*>' is invalid: there is no unique or primary key constraint on table '<*tableName*>' that matches the number and types of the columns in the foreign key.

SQLSTATE	Message Text
X0Y45	Foreign key constraint '<*constraintName*>' cannot be added to or enabled on table <*tableName*> because one or more foreign keys do not have matching referenced keys.
X0Y46	Constraint '<*constraintName*>' is invalid: referenced table <*tableName*> does not exist.
X0Y54	Schema '<*schemaNamet*>' cannot be dropped because it is not empty.
X0Y55	The number of rows in the base table does not match the number of rows in at least 1 of the indexes on the table. Index '<*indexName*>' on table '<*schemaNamet*>.<*tableName*>' has <*number*> rows, but the base table has <*number*> rows. The suggested corrective action is to recreate the index.
X0Y56	'<*value*>' is not allowed on the System table '<*tableName*>'.
X0Y57	A non-nullable column cannot be added to table '<*tableName*>' because the table contains at least one row. Non-nullable columns can only be added to empty tables.
X0Y58	Attempt to add a primary key constraint to table '<*tableName*>' failed because the table already has a constraint of that type. A table can only have a single primary key constraint.
X0Y59	Attempt to add or enable constraint(s) on table '<*rowName*>' failed because the table contains <*constraintName*> row(s) that violate the following check constraint(s): <*tableName*>.
X0Y63	The command on table '<*tableName*>' failed because null data was found in the primary key or unique constraint/index column(s). All columns in a primary or unique index key must not be null.
X0Y63	The command on table '<*tableName*>' failed because null data was found in the primary key/index column(s). All columns in a primary key must not be null.
X0Y66	Cannot issue commit in a nested connection when there is a pending operation in the parent connection.
X0Y67	Cannot issue rollback in a nested connection when there is a pending operation in the parent connection.
X0Y68	<*value*> '<*value*>' already exists.
X0Y69	<*triggerName*> is not supported in trigger <*value*>.
X0Y70	INSERT, UPDATE and DELETE are not permitted on table <*triggerName*> because trigger <*tableName*> is active.
X0Y71	Transaction manipulation such as SET ISOLATION is not permitted because trigger <*triggerName*> is active.
X0Y72	Bulk insert replace is not permitted on '<*value*>' because it has an enabled trigger (<*value*>).
X0Y77	Cannot issue set transaction isolation statement on a global transaction that is in progress because it would have implicitly committed the global transaction.
X0Y78	Statement.executeQuery() cannot be called with a statement that returns a row count.

SQLSTATE	Message Text
X0Y78	<*value*>.executeQuery() cannot be called because multiple result sets were returned. Use <*value*>.execute() to obtain multiple results.
X0Y78	<*value*>.executeQuery() was called but no result set was returned. Use <*value*>.executeUpdate() for non-queries.
X0Y79	Statement.executeUpdate() cannot be called with a statement that returns a ResultSet.
X0Y80	ALTER table '<*tableName*>' failed. Null data found in column '<*columnName*>'.
X0Y83	WARNING: While deleting a row from a table the index row for base table row <*rowName*> was not found in index with conglomerate id <*id*>. This problem has automatically been corrected as part of the delete operation.

Table 60. Class XBCA: CacheService

SQLSTATE	Message Text
XBCA0	Cannot create new object with key <*cache*> in <*key*> cache. The object already exists in the cache.

Table 61. Class XBCM: ClassManager

SQLSTATE	Message Text
XBCM1	Java linkage error thrown during load of generated class <*className*>.
XBCM2	Cannot create an instance of generated class <*className*>.
XBCM3	Method <*className*>() does not exist in generated class <*methodName*>.
XBCM4	Java class file format limit(s) exceeded: <*className*> in generated class <*value*>.

Table 62. Class XBCX: Cryptography

SQLSTATE	Message Text
XBCX0	Exception from Cryptography provider. See next exception for details.
XBCX1	Initializing cipher with illegal mode, must be either ENCRYPT or DECRYPT.
XBCX2	Initializing cipher with a boot password that is too short. The password must be at least <*number*> characters long.
XBCX5	Cannot change boot password to null.
XBCX6	Cannot change boot password to a non-string serializable type.
XBCX7	Wrong format for changing boot password. Format must be : old_boot_password, new_boot_password.
XBCX8	Cannot change boot password for a non-encrypted database.
XBCX9	Cannot change boot password for a read-only database.
XBCXA	Wrong boot password.
XBCXB	Bad encryption padding '<*value*>' or padding not specified. 'NoPadding' must be used.

SQLSTATE	Message Text
XBCXC	Encryption algorithm '<*algorithmName*>' does not exist. Please check that the chosen provider '<*providerName*>' supports this algorithm.
XBCXD	The encryption algorithm cannot be changed after the database is created.
XBCXE	The encryption provider cannot be changed after the database is created.
XBCXF	The class '<*className*>' representing the encryption provider cannot be found.
XBCXG	The encryption provider '<*providerName*>' does not exist.
XBCXH	The encryptionAlgorithm '<*algorithmName*>' is not in the correct format. The correct format is algorithm/feedbackMode/NoPadding.
XBCXI	The feedback mode '<*mode*>' is not supported. Supported feedback modes are CBC, CFB, OFB and ECB.
XBCXJ	The application is using a version of the Java Cryptography Extension (JCE) earlier than 1.2.1. Please upgrade to JCE 1.2.1 and try the operation again.
XBCXK	The given encryption key does not match the encryption key used when creating the database. Please ensure that you are using the correct encryption key and try again.
XBCXL	The verification process for the encryption key was not successful. This could have been caused by an error when accessing the appropriate file to do the verification process. See next exception for details.
XBCXM	The length of the external encryption key must be an even number.
XBCXN	The external encryption key contains one or more illegal characters. Allowed characters for a hexadecimal number are 0-9, a-f and A-F.
XBCXO	Cannot encrypt the database when there is a global transaction in the prepared state.
XBCXP	Cannot re-encrypt the database with a new boot password or an external encryption key when there is a global transaction in the prepared state.
XBCXQ	Cannot configure a read-only database for encryption.
XBCXR	Cannot re-encrypt a read-only database with a new boot password or an external encryption key .
XBCXS	Cannot configure a database for encryption, when database is in the log archive mode.
XBCXT	Cannot re-encrypt a database with a new boot password or an external encryption key, when database is in the log archive mode.
XBCXU	Encryption of an un-encrypted database failed: <*failureMessage*>
XBCXV	Encryption of an encrypted database with a new key or a new password failed: <*failureMessage*>

Table 63. Class XBM: Monitor

SQLSTATE	Message Text
XBM01	Startup failed due to an exception. See next exception for details.
XBM02	Startup failed due to missing functionality for <*value*>. Please ensure your classpath includes the correct Derby software.

SQLSTATE	Message Text
XBM05	Startup failed due to missing product version information for <*value*>.
XBM06	Startup failed. An encrypted database cannot be accessed without the correct boot password.
XBM07	Startup failed. Boot password must be at least 8 bytes long.
XBM08	Could not instantiate <*value*> StorageFactory class <*value*>.
XBM0G	Failed to start encryption engine. Please make sure you are running Java 2 and have downloaded an encryption provider such as jce and put it in your class path.
XBM0H	Directory <*directoryName*> cannot be created.
XBM0I	Directory <*directoryName*> cannot be removed.
XBM0J	Directory <*directoryName*> already exists.
XBM0K	Unknown sub-protocol for database name <*databaseName*>.
XBM0L	Specified authentication scheme class <*className*> does implement the authentication interface <*interfaceName*>.
XBM0M	Error creating instance of authentication scheme class <*className*>.
XBM0N	JDBC Driver registration with java.sql.DriverManager failed. See next exception for details.
XBM0P	Service provider is read-only. Operation not permitted.
XBM0Q	File <*fileName*> not found. Please make sure that backup copy is the correct one and it is not corrupted.
XBM0R	Unable to remove File <*fileName*>.
XBM0S	Unable to rename file '<*fileName*>' to '<*fileName*>'
XBM0T	Ambiguous sub-protocol for database name <*databaseName*>.
XBM0U	No class was registered for identifier <*identifierName*>.
XBM0V	An exception was thrown while loading class <*identifierName*> registered for identifier <*className*>.
XBM0W	An exception was thrown while creating an instance of class <*identifierName*> registered for identifier <*className*>.
XBM0X	Supplied territory description '<*value*>' is invalid, expecting ln[_CO[_variant]] ln=lower-case two-letter ISO-639 language code, CO=upper-case two-letter ISO-3166 country codes, see java.util.Locale.
XBM03	Supplied value '<*value*>' for collation attribute is invalid, expecting UCS_BASIC or TERRITORY_BASED.
XBM04	Collator support not available from the JVM for the database's locale '<*value*>'.
XBM0Y	Backup database directory <*directoryName*> not found. Please make sure that the specified backup path is right.
XBM0Z	Unable to copy file '<*fileName*>' to '<*fileName*>'. Please make sure that there is enough space and permissions are correct.

Table 64. Class XCL: Execution exceptions

SQLSTATE	Message Text
XCL01	Result set does not return rows. Operation <*operationName*> not permitted.
XCL05	Activation closed, operation <*operationName*> not permitted.
XCL07	Cursor '<*cursorName*>' is closed. Verify that autocommit is OFF.
XCL08	Cursor '<*cursorName*>' is not on a row.
XCL09	An Activation was passed to the '<*methodName*>' method that does not match the PreparedStatement.
XCL10	A PreparedStatement has been recompiled and the parameters have changed. If you are using JDBC you must prepare the statement again.
XCL12	An attempt was made to put a data value of type '<*datatypeName*>' into a data value of type '<*datatypeName*>'.
XCL13	The parameter position '<*parameterPosition*>' is out of range. The number of parameters for this prepared statement is '<*number*>'.
XCL14	The column position '<*columnPosition*>' is out of range. The number of columns for this ResultSet is '<*number*>'.
XCL15	A ClassCastException occurred when calling the compareTo() method on an object '<*object*>'. The parameter to compareTo() is of class '<*className*>'.
XCL16	ResultSet not open. Operation '<*operation*>' not permitted. Verify that autocommit is OFF.
XCL16	ResultSet not open. Verify that autocommit is OFF.
XCL18	Stream of column value in result cannot be retrieved twice
XCL19	Missing row in table '<*tableName*>' for key '<*key*>'.
XCL20	Catalogs at version level '<*versionNumber*>' cannot be upgraded to version level '<*versionNumber*>'.
XCL21	You are trying to execute a Data Definition statement (CREATE, DROP, or ALTER) while preparing a different statement. This is not allowed. It can happen if you execute a Data Definition statement from within a static initializer of a Java class that is being used from within a SQL statement.
XCL22	Parameter <*parameterName*> cannot be registered as an OUT parameter because it is an IN parameter.
XCL23	SQL type number '<*type*>' is not a supported type by registerOutParameter().
XCL24	Parameter <*parameterName*> appears to be an output parameter, but it has not been so designated by registerOutParameter(). If it is not an output parameter, then it has to be set to type <*type*>.
XCL25	Parameter <*parameterName*> cannot be registered to be of type <*type*> because it maps to type <*type*> and they are incompatible.
XCL26	Parameter <*parameterName*> is not an output parameter.
XCL27	Return output parameters cannot be set.
XCL30	An IOException was thrown when reading a '<*value*>' from an InputStream.
XCL31	Statement closed.

SQLSTATE	Message Text
XCL33	The table cannot be defined as a dependent of table <*tableName*> because of delete rule restrictions. (The relationship is self-referencing and a self-referencing relationship already exists with the SET NULL delete rule.)
XCL34	The table cannot be defined as a dependent of table <*tableName*> because of delete rule restrictions. (The relationship forms a cycle of two or more tables that cause the table to be delete-connected to itself (all other delete rules in the cycle would be CASCADE)).
XCL35	The table cannot be defined as a dependent of table <*tableName*> because of delete rule restrictions. (The relationship causes the table to be delete-connected to the indicated table through multiple relationships and the delete rule of the existing relationship is SET NULL.).
XCL36	The delete rule of foreign key must be <*value*>. (The referential constraint is self-referencing and an existing self-referencing constraint has the indicated delete rule (NO ACTION, RESTRICT or CASCADE).)
XCL37	The delete rule of foreign key must be <*value*>. (The referential constraint is self-referencing and the table is dependent in a relationship with a delete rule of CASCADE.)
XCL38	the delete rule of foreign key must be <*ruleName*>. (The relationship would cause the table to be delete-connected to the same table through multiple relationships and such relationships must have the same delete rule (NO ACTION, RESTRICT or CASCADE).)
XCL39	The delete rule of foreign key cannot be CASCADE. (A self-referencing constraint exists with a delete rule of SET NULL, NO ACTION or RESTRICT.)
XCL40	The delete rule of foreign key cannot be CASCADE. (The relationship would form a cycle that would cause a table to be delete-connected to itself. One of the existing delete rules in the cycle is not CASCADE, so this relationship may be definable if the delete rule is not CASCADE.)
XCL41	the delete rule of foreign key can not be CASCADE. (The relationship would cause another table to be delete-connected to the same table through multiple paths with different delete rules or with delete rule equal to SET NULL.)
XCL42	CASCADE
XCL43	SET NULL
XCL44	RESTRICT
XCL45	NO ACTION
XCL46	SET DEFAULT
XCL47	Use of '<*value*>' requires database to be upgraded from version <*versionNumber*> to version <*versionNumber*> or later.
XCL48	TRUNCATE TABLE is not permitted on '<*value*>' because unique/primary key constraints on this table are referenced by enabled foreign key constraints from other tables.
XCL49	TRUNCATE TABLE is not permitted on '<*value*>' because it has an enabled DELETE trigger (<*value*>).

SQLSTATE	Message Text
XCL50	Upgrading the database from a previous version is not supported. The database being accessed is at version level '<*versionNumber*>', this software is at version level '<*versionNumber*>'.
XCL51	The requested function can not reference tables in SESSION schema.
XCL52	The statement has been cancelled or timed out.
XCL53	Stream is closed

Table 65. Class XCW: Upgrade unsupported

SQLSTATE	Message Text
XCW00	Unsupported upgrade from '<*value*>' to '<*value*>'.

Table 66. Class XCX: Internal Utility Errors

SQLSTATE	Message Text
XCXA0	Invalid identifier.
XCXB0	Invalid database classpath: '<*classpath*>'.
XCXC0	Invalid id list.
XCXE0	You are trying to do an operation that uses the territory of the database, but the database does not have a territory.

Table 67. Class XCY: Derby Property Exceptions

SQLSTATE	Message Text
XCY00	Invalid value for property '<*value*>'='<*value*>'.
XCY02	The requested property change is not supported '<*value*>'='<*value*>'.
XCY03	Required property '<*propertyName*>' has not been set.
XCY04	Invalid syntax for optimizer overrides. The syntax should be -- DERBY-PROPERTIES propertyName = value [, propertyName = value]*

Table 68. Class XCZ: org.apache.derby.database.UserUtility

SQLSTATE	Message Text
XCZ00	Unknown permission '<*permissionName*>'.
XCZ01	Unknown user '<*authorizationID*>'.
XCZ02	Invalid parameter '<*value*>'='<*value*>'.

Table 69. Class XD00: Dependency Manager

SQLSTATE	Message Text
XD003	Unable to restore dependency from disk. DependableFinder = '<*value*>'. Further information: '<*value*>'.

SQLSTATE	Message Text
XD004	Unable to store dependencies.

Table 70. Class XIE: Import/Export Exceptions

SQLSTATE	Message Text
XIE01	Connection was null.
XIE03	Data found on line <*lineNumber*> for column <*columnName*> after the stop delimiter.
XIE04	Data file not found: <*fileName*>
XIE05	Data file cannot be null.
XIE06	Entity name was null.
XIE07	Field and record separators cannot be substrings of each other.
XIE08	There is no column named: <*columnName*>.
XIE09	The total number of columns in the row is: <*number*>.
XIE0B	Column '<*columnName*>' in the table is of type <*type*>, it is not supported by the import/export feature.
XIE0D	Cannot find the record separator on line <*lineNumber*>.
XIE0E	Read endOfFile at unexpected place on line <*lineNumber*>.
XIE0I	An IOException occurred while writing data to the file.
XIE0J	A delimiter is not valid or is used more than once.
XIE0K	The period was specified as a character string delimiter.
XIE0M	Table '<*tableName*>' does not exist.
XIE0N	An invalid hexadecimal string '<*hexString*>' detected in the import file.
XIE0P	Lob data file <*fileName*> referenced in the import file not found.
XIE0Q	Lob data file name cannot be null.
XIE0R	Import error on line <*lineNumber*> of file <*fileName*>: <*details*>
XIE0S	The export operation was not performed, because the specified output file (<*fileName*>) already exists. Export processing will not overwrite an existing file, even if the process has permissions to write to that file, due to security concerns, and to avoid accidental file damage. Please either change the output file name in the export procedure arguments to specify a file which does not exist, or delete the existing file, then retry the export operation.
XIE0T	The export operation was not performed, because the specified large object auxiliary file (<*fileName*>) already exists. Export processing will not overwrite an existing file, even if the process has permissions to write to that file, due to security concerns, and to avoid accidental file damage. Please either change the large object auxiliary file name in the export procedure arguments to specify a file which does not exist, or delete the existing file, then retry the export operation.

Table 71. Class XJ: Connectivity Errors

SQLSTATE	Message Text
XJ004	Database '<*databaseName*>' not found.
XJ008	Cannot rollback or release a savepoint when in auto-commit mode.
XJ009	Use of CallableStatement required for stored procedure call or use of output parameters: <*value*>
XJ010	Cannot issue savepoint when autoCommit is on.
XJ011	Cannot pass null for savepoint name.
XJ012	'<*value*>' already closed.
XJ013	No ID for named savepoints.
XJ014	No name for un-named savepoints.
XJ015	Derby system shutdown.
XJ016	Method '<*methodName*>' not allowed on prepared statement.
XJ017	No savepoint command allowed inside the trigger code.
XJ018	Column name cannot be null.
XJ020	Object type not convertible to TYPE '<*typeName*>', invalid java.sql.Types value, or object was null.
XJ021	Type is not supported.
XJ022	Unable to set stream: '<*name*>'.
XJ023	Input stream did not have exact amount of data as the requested length.
XJ025	Input stream cannot have negative length.
XJ028	The URL '<*urlValue*>' is not properly formed.
XJ030	Cannot set AUTOCOMMIT ON when in a nested connection.
XJ040	Failed to start database '<*databaseName*>', see the next exception for details.
XJ041	Failed to create database '<*databaseName*>', see the next exception for details.
XJ042	'<*value*>' is not a valid value for property '<*propertyName*>'.
XJ044	'<*value*>' is an invalid scale.
XJ045	Invalid or (currently) unsupported isolation level, '<*levelName*>', passed to Connection.setTransactionIsolation(). The currently supported values are java.sql.Connection.TRANSACTION_SERIALIZABLE, java.sql.Connection.TRANSACTION_REPEATABLE_READ, java.sql.Connection.TRANSACTION_READ_COMMITTED, and java.sql.Connection.TRANSACTION_READ_UNCOMMITTED.
XJ049	Conflicting create attributes specified.
XJ04B	Batch cannot contain a command that attempts to return a result set.
XJ04C	CallableStatement batch cannot contain output parameters.
XJ056	Cannot set AUTOCOMMIT ON when in an XA connection.
XJ057	Cannot commit a global transaction using the Connection, commit processing must go thru XAResource interface.

SQLSTATE	Message Text
XJ058	Cannot rollback a global transaction using the Connection, commit processing must go thru XAResource interface.
XJ059	Cannot close a connection while a global transaction is still active.
XJ05B	JDBC attribute '*<attributeName>*' has an invalid value '*<value>*', valid values are '*<value>*'.
XJ05C	Cannot set holdability ResultSet.HOLD_CURSORS_OVER_COMMIT for a global transaction.
XJ061	The '*<methodName>*' method is only allowed on scroll cursors.
XJ062	Invalid parameter value '*<value>*' for ResultSet.setFetchSize(int rows).
XJ063	Invalid parameter value '*<value>*' for Statement.setMaxRows(int maxRows). Parameter value must be >= 0.
XJ064	Invalid parameter value '*<value>*' for setFetchDirection(int direction).
XJ065	Invalid parameter value '*<value>*' for Statement.setFetchSize(int rows).
XJ066	Invalid parameter value '*<value>*' for Statement.setMaxFieldSize(int max).
XJ067	SQL text pointer is null.
XJ068	Only executeBatch and clearBatch allowed in the middle of a batch.
XJ069	No SetXXX methods allowed in case of USING execute statement.
XJ070	Negative or zero position argument '*<argument>*' passed in a Blob or Clob method.
XJ071	Negative length argument '*<argument>*' passed in a BLOB or CLOB method.
XJ072	Null pattern or searchStr passed in to a BLOB or CLOB position method.
XJ073	The data in this BLOB or CLOB is no longer available. The BLOB/CLOB's transaction may be committed, its connection closed or it has been freed.
XJ074	Invalid parameter value '*<value>*' for Statement.setQueryTimeout(int seconds).
XJ076	The position argument '*<positionArgument>*' exceeds the size of the BLOB/CLOB.
XJ077	Got an exception when trying to read the first byte/character of the BLOB/CLOB pattern using getBytes/getSubString.
XJ078	Offset '*<value>*' is either less than zero or is too large for the current BLOB/CLOB.
XJ079	The length specified '*<number>*' exceeds the size of the BLOB/CLOB.
XJ080	USING execute statement passed *<number>* parameters rather than *<number>*.
XJ081	Conflicting create/restore/recovery attributes specified.
XJ081	Invalid value '*<value>*' passed as parameter '*<parameterName>*' to method '*<methodName>*'
XJ085	Stream has already been read and end-of-file reached and cannot be re-used.
XJ086	This method cannot be invoked while the cursor is not on the insert row or if the concurrency of this ResultSet object is CONCUR_READ_ONLY.

SQLSTATE	Message Text
XJ087	Sum of position('<*pos*>') and length('<*length*>') is greater than the size of the LOB plus one.
XJ088	Invalid operation: wasNull() called with no data retrieved.
XJ090	Invalid parameter: calendar is null.
XJ091	Invalid argument: parameter index <*indexNumber*> is not an OUT or INOUT parameter.
XJ093	Length of BLOB/CLOB, <*number*>, is too large. The length cannot exceed <*number*>.
XJ094	This object is already closed.
XJ095	An attempt to execute a privileged action failed.
XJ096	A resource bundle could not be found in the <*packageName*> package for <*value*>
XJ097	Cannot rollback or release a savepoint that was not created by this connection.
XJ098	The auto-generated keys value <*value*> is invalid
XJ099	The Reader/Stream object does not contain length characters
XJ100	The scale supplied by the registerOutParameter method does not match with the setter method. Possible loss of precision!
XJ103	Table name can not be null
XJ104	Shared key length is invalid: <*value*>.
XJ105	DES key has the wrong length, expected length <*number*>, got length <*number*>.
XJ106	No such padding
XJ107	Bad Padding
XJ108	Illegal Block Size
XJ110	Primary table name can not be null
XJ111	Foreign table name can not be null
XJ112	Security exception encountered, see next exception for details.
XJ113	Unable to open file <*fileName*> : <*error*>
XJ114	Invalid cursor name '<*cursorName*>'
XJ115	Unable to open resultSet with requested holdability <*value*>.
XJ116	No more than <*number*> commands may be added to a single batch.
XJ117	Batching of queries not allowed by J2EE compliance.
XJ118	Query batch requested on a non-query statement.
XJ121	Invalid operation at current cursor position.
XJ122	No updateXXX methods were called on this row.
XJ123	This method must be called to update values in the current row or the insert row.
XJ124	Column not updatable.

SQLSTATE	Message Text
XJ125	This method should only be called on ResultSet objects that are scrollable (type TYPE_SCROLL_INSENSITIVE).
XJ126	This method should not be called on sensitive dynamic cursors.
XJ128	Unable to unwrap for '<*value*>'
XJ200	Exceeded maximum number of sections <*value*>
XJ202	Invalid cursor name '<*cursorName*>'.
XJ203	Cursor name '<*cursorName*>' is already in use
XJ204	Unable to open result set with requested holdability <*holdValue*>.
XJ206	SQL text '<*value*>' has no tokens.
XJ207	executeQuery method can not be used for update.
XJ208	Non-atomic batch failure. The batch was submitted, but at least one exception occurred on an individual member of the batch. Use getNextException() to retrieve the exceptions for specific batched elements.
XJ209	The required stored procedure is not installed on the server.
XJ210	The load module name for the stored procedure on the server is not found.
XJ211	Non-recoverable chain-breaking exception occurred during batch processing. The batch is terminated non-atomically.
XJ212	Invalid attribute syntax: <*attributeSyntax*>
XJ213	The traceLevel connection property does not have a valid format for a number.
XJ214	An IO Error occurred when calling free() on a CLOB or BLOB.
XJ215	You cannot invoke other java.sql.Clob/java.sql.Blob methods after calling the free() method or after the Blob/Clob's transaction has been committed or rolled back.
XJ216	The length of this BLOB/CLOB is not available yet. When a BLOB or CLOB is accessed as a stream, the length is not available until the entire stream has been processed.
XJ217	The locator that was supplied for this CLOB/BLOB is invalid

Table 72. Class XK: Security Exceptions

SQLSTATE	Message Text
XK000	The security policy could not be reloaded: <*reason*>

Table 73. Class XN: Network Client Exceptions

SQLSTATE	Message Text
XN001	Connection reset is not allowed when inside a unit of work.
XN008	Query processing has been terminated due to an error on the server.
XN009	Error obtaining length of BLOB/CLOB object, exception follows.
XN010	Procedure name can not be null.

SQLSTATE	Message Text
XN011	Procedure name length *<number>* is not within the valid range of 1 to *<number>*.
XN012	On *<operatingSystemName>* platforms, XA supports version *<versionNumber>* and above, this is version *<versionNumber>*
XN013	Invalid scroll orientation.
XN014	Network protocol error: encountered an IOException, parameter #*<value>*. Remaining data has been padded with 0x0. Message: *<messageText>*.
XN015	Network protocol error: the specified size of the InputStream, parameter #*<value>*, is less than the actual InputStream length.
XN016	Network protocol error: encountered error in stream length verification, parameter #*<value>*. Message: *<messageText>*.
XN017	Network protocol error: end of stream prematurely reached, parameter #*<value>*. Remaining data has been padded with 0x0.
XN018	Network protocol error: the specified size of the Reader, parameter #*<value>*, is less than the actual InputStream length.
XN019	Error executing a *<value>*, server returned *<value>*.

Table 74. Class XRE: Replication Exceptions

SQLSTATE	Message Text
XRE00	This LogFactory module does not support replicatiosn.
XRE01	The log received from the master is corrupted.
XRE02	Master and Slave at different versions. Unable to proceed with Replication.
XRE03	Unexpected replication error. See derby.log for details.
XRE04	Could not establish a connection to the peer of the replicated database '*<dbname>*' on address '*<hostname>*:*<portname>*'.
XRE04	Connection lost for replicated database '*<dbname>*'.
XRE05	The log files on the master and slave are not in synch for replicated database '*<dbname>*'. The master log instant is *<masterfile>*:*<masteroffset>*, whereas the slave log instant is *<slavefile>*:*<slaveoffset>*. This is FATAL for replication - replication will be stopped.
XRE06	The connection attempts to the replication slave for the database *<dbname>* exceeded the specified timeout period.
XRE07	Could not perform operation because the database is not in replication master mode.
XRE08	Replication slave mode started successfully for database '*<dbname>*'. Connection refused because the database is in replication slave mode.
XRE09	Cannot start replication slave mode for database '*<dbname>*'. The database has already been booted.
XRE10	Conflicting attributes specified. See reference manual for attributes allowed in combination with replication attribute '*<attribute>*'.
XRE11	Could not perform operation '*<command>*' because the database '*<dbname>*' has not been booted.

SQLSTATE	Message Text
XRE12	Replication network protocol error for database '<*dbname*>'. Expected message type '<*expectedtype*>', but received type '<*receivedtype*>'.
XRE20	Failover performed successfully for database '<*dbname*>', the database has been shutdown.
XRE21	Error occurred while performing failover for database '<*dbname*>', Failover attempt was aborted.
XRE22	Replication master has already been booted for database '<*dbname*>'
XRE23	Replication master cannot be started since unlogged operations are in progress, unfreeze to allow unlogged operations to complete and restart replication
XRE40	Could not perform operation because the database is not in replication slave mode.
XRE41	Replication operation 'failover' or 'stopSlave' refused on the slave database because the connection with the master is working. Issue the 'failover' or 'stopMaster' operation on the master database instead.
XRE42	Replicated database '<*dbname*>' shutdown.
XRE43	Unexpected error when trying to stop replication slave mode. To stop repliation slave mode, use operation 'stopSlave' or 'failover'.

Table 75. Class XSAI: Store - access.protocol.interface

SQLSTATE	Message Text
XSAI2	The conglomerate (<*value*>) requested does not exist.
XSAI3	Feature not implemented.

Table 76. Class XSAM: Store - AccessManager

SQLSTATE	Message Text
XSAM0	Exception encountered while trying to boot module for '<*value*>'.
XSAM2	There is no index or conglomerate with conglom id '<*conglomID*>' to drop.
XSAM3	There is no index or conglomerate with conglom id '<*conglomID*>'.
XSAM4	There is no sort called '<*sortName*>'.
XSAM5	Scan must be opened and positioned by calling next() before making other calls.
XSAM6	Record <*containerName*> on page <*pageNumber*> in container <*recordNumber*> not found.

Table 77. Class XSAS: Store - Sort

SQLSTATE	Message Text
XSAS0	A scan controller interface method was called which is not appropriate for a scan on a sort.

SQLSTATE	Message Text
XSAS1	An attempt was made to fetch a row before the beginning of a sort or after the end of a sort.
XSAS3	The type of a row inserted into a sort does not match the sort's template.
XSAS6	Could not acquire resources for sort.

Table 78. Class XSAX: Store - access.protocol.XA statement

SQLSTATE	Message Text
XSAX0	XA protocol violation.
XSAX1	An attempt was made to start a global transaction with an Xid of an existing global transaction.

Table 79. Class XSCB: Store - BTree

SQLSTATE	Message Text
XSCB0	Could not create container.
XSCB1	Container <*containerName*> not found.
XSCB2	The required property <*propertyName*> not found in the property list given to createConglomerate() for a btree secondary index.
XSCB3	Unimplemented feature.
XSCB4	A method on a btree open scan has been called prior to positioning the scan on the first row (i.e. no next() call has been made yet). The current state of the scan is (<*value*>).
XSCB5	During logical undo of a btree insert or delete the row could not be found in the tree.
XSCB6	Limitation: Record of a btree secondary index cannot be updated or inserted due to lack of space on the page. Use the parameters derby.storage.pageSize and/or derby.storage.pageReservedSpace to work around this limitation.
XSCB7	An internal error was encountered during a btree scan - current_rh is null = <*value*>, position key is null = <*value*>.
XSCB8	The btree conglomerate <*value*> is closed.
XSCB9	Reserved for testing.

Table 80. Class XSCG0: Conglomerate

SQLSTATE	Message Text
XSCG0	Could not create a template.

Table 81. Class XSCH: Heap

SQLSTATE	Message Text
XSCH0	Could not create container.

SQLSTATE	Message Text
XSCH1	Container <*containerName*> not found.
XSCH4	Conglomerate could not be created.
XSCH5	In a base table there was a mismatch between the requested column number <*number*> and the maximum number of columns <*number*>.
XSCH6	The heap container with container id <*containerID*> is closed.
XSCH7	The scan is not positioned.
XSCH8	The feature is not implemented.

Table 82. Class XSDA: RawStore - Data.Generic statement

SQLSTATE	Message Text
XSDA1	An attempt was made to access an out of range slot on a page
XSDA2	An attempt was made to update a deleted record
XSDA3	Limitation: Record cannot be updated or inserted due to lack of space on the page. Use the parameters derby.storage.pageSize and/or derby.storage.pageReservedSpace to work around this limitation.
XSDA4	An unexpected exception was thrown
XSDA5	An attempt was made to undelete a record that is not deleted
XSDA6	Column <*columnName*> of row is null, it needs to be set to point to an object.
XSDA7	Restore of a serializable or SQLData object of class <*className*>, attempted to read more data than was originally stored
XSDA8	Exception during restore of a serializable or SQLData object of class <*className*>
XSDA9	Class not found during restore of a serializable or SQLData object of class <*className*>
XSDAA	Illegal time stamp <*value*>, either time stamp is from a different page or of incompatible implementation
XSDAB	cannot set a null time stamp
XSDAC	Attempt to move either rows or pages from one container to another.
XSDAD	Attempt to move zero rows from one page to another.
XSDAE	Can only make a record handle for special record handle id.
XSDAF	Using special record handle as if it were a normal record handle.
XSDAG	The allocation nested top transaction cannot open the container.
XSDAI	Page <*page*> being removed is already locked for deallocation.
XSDAJ	Exception during write of a serializable or SQLData object
XSDAK	Wrong page is gotten for record handle <*value*>.
XSDAL	Record handle <*value*> unexpectedly points to overflow page.
XSDAM	Exception during restore of a SQLData object of class <*className*>. The specified class cannot be instantiated.
XSDAN	

SQLSTATE	Message Text
	Exception during restore of a SQLData object of class <*className*>. The specified class encountered an illegal access exception.

Table 83. Class XSDB: RawStore - Data.Generic transaction

SQLSTATE	Message Text
XSDB0	Unexpected exception on in-memory page <*page*>
XSDB1	Unknown page format at page <*page*>
XSDB2	Unknown container format at container <*containerName*> : <*value*>
XSDB3	Container information cannot change once written: was <*value*>, now <*value*>
XSDB4	Page <*page*> is at version <*versionNumber*>, the log file contains change version <*versionNumber*>, either there are log records of this page missing, or this page did not get written out to disk properly.
XSDB5	Log has change record on page <*page*>, which is beyond the end of the container.
XSDB6	Another instance of Derby may have already booted the database <*databaseName*>.
XSDB7	WARNING: Derby (instance <*value*>) is attempting to boot the database <*databaseName*> even though Derby (instance <*value*>) may still be active. Only one instance of Derby should boot a database at a time. Severe and non-recoverable corruption can result and may have already occurred.
XSDB8	WARNING: Derby (instance <*value*>) is attempting to boot the database <*databaseName*> even though Derby (instance <*value*>) may still be active. Only one instance of Derby should boot a database at a time. Severe and non-recoverable corruption can result if 2 instances of Derby boot on the same database at the same time. The db2j.database.forceDatabaseLock=true property has been set, so the database will not boot until the db.lck is no longer present. Normally this file is removed when the first instance of Derby to boot on the database exits, but it may be left behind in some shutdowns. It will be necessary to remove the file by hand in that case. It is important to verify that no other VM is accessing the database before deleting the db.lck file by hand.
XSDB9	Stream container <*containerName*> is corrupt.
XSDBA	Attempt to allocate object <*object*> failed.
XSDBB	Unknown page format at page <*page*>, page dump follows: <*value*>

Table 84. Class XSDF: RawStore - Data.Filesystem statement

SQLSTATE	Message Text
XSDF0	Could not create file <*fileName*> as it already exists.
XSDF1	Exception during creation of file <*fileName*> for container
XSDF2	Exception during creation of file <*fileName*> for container, file could not be removed. The exception was: <*value*>.
XSDF3	Cannot create segment <*segmentName*>.

SQLSTATE	Message Text
XSDF4	Exception during remove of file <*fileName*> for dropped container, file could not be removed <*value*>.
XSDF6	Cannot find the allocation page <*page*>.
XSDF7	Newly created page failed to be latched <*value*>
XSDF8	Cannot find page <*page*> to reuse.
XSDFB	Operation not supported by a read only database
XSDFD	Different page image read on 2 I/Os on Page <*page*>, first image has incorrect checksum, second image has correct checksum. Page images follows: <*value*><*value*>
XSDFF	The requested operation failed due to an unexpected exception.
XSDFH	Cannot backup the database, got an I/O Exception while writing to the backup container file <*fileName*>.
XSDFI	Error encountered while trying to write data to disk during database recovery. Check that the database disk is not full. If it is then delete unnecessary files, and retry connecting to the database. It is also possible that the file system is read only, or the disk has failed, or some other problem with the media. System encountered error while processing page <*page*>.

Table 85. Class XSDG: RawStore - Data.Filesystem database

SQLSTATE	Message Text
XSDG0	Page <*page*> could not be read from disk.
XSDG1	Page <*page*> could not be written to disk, please check if disk is full.
XSDG2	Invalid checksum on Page <*page*>, expected=<*value*>, on-disk version=<*value*>, page dump follows: <*value*>
XSDG3	Meta-data for Container <*containerName*> could not be accessed
XSDG5	Database is not in create mode when createFinished is called.
XSDG6	Data segment directory not found in <*value*> backup during restore. Please make sure that backup copy is the right one and it is not corrupted.
XSDG7	Directory <*directoryName*> could not be removed during restore. Please make sure that permissions are correct.
XSDG8	Unable to copy directory '<*directoryName*>' to '<*value*>' during restore. Please make sure that there is enough space and permissions are correct.

Table 86. Class XSLA: RawStore - Log.Generic database exceptions

SQLSTATE	Message Text
XSLA0	Cannot flush the log file to disk <*value*>.
XSLA1	Log Record has been sent to the stream, but it cannot be applied to the store (Object <*object*>). This may cause recovery problems also.
XSLA2	System will shutdown, got I/O Exception while accessing log file.
XSLA3	Log Corrupted, has invalid data in the log stream.

SQLSTATE	Message Text
XSLA4	Cannot write to the log, most likely the log is full. Please delete unnecessary files. It is also possible that the file system is read only, or the disk has failed, or some other problems with the media.
XSLA5	Cannot read log stream for some reason to rollback transaction <*transactionID*>.
XSLA6	Cannot recover the database.
XSLA7	Cannot redo operation <*operation*> in the log.
XSLA8	Cannot rollback transaction <*value*>, trying to compensate <*value*> operation with <*value*>
XSLAA	The store has been marked for shutdown by an earlier exception.
XSLAB	Cannot find log file <*logfileName*>, please make sure your logDevice property is properly set with the correct path separator for your platform.
XSLAC	Database at <*value*> have incompatible format with the current version of software, it may have been created by or upgraded by a later version.
XSLAD	log Record at instant <*value*> in log file <*value*> corrupted. Expected log record length <*value*>, real length <*logfileName*>.
XSLAE	Control file at <*value*> cannot be written or updated.
XSLAF	A Read Only database was created with dirty data buffers.
XSLAH	A Read Only database is being updated.
XSLAI	Cannot log the checkpoint log record
XSLAJ	The logging system has been marked to shut down due to an earlier problem and will not allow any more operations until the system shuts down and restarts.
XSLAK	Database has exceeded largest log file number <*value*>.
XSLAL	log record size <*logfileName*> exceeded the maximum allowable log file size <*value*>. Error encountered in log file <*value*>, position <*number*>.
XSLAM	Cannot verify database format at {1} due to IOException.
XSLAN	Database at <*value*> has an incompatible format with the current version of the software. The database was created by or upgraded by version <*versionNumber*>.
XSLAO	Recovery failed unexpected problem <*value*>.
XSLAP	Database at <*value*> is at version <*versionNumber*>. Beta databases cannot be upgraded,
XSLAQ	cannot create log file at directory <*directoryName*>.
XSLAR	Unable to copy log file '<*logfileName*>' to '<*value*>' during restore. Please make sure that there is enough space and permissions are correct.
XSLAS	Log directory <*directoryName*> not found in backup during restore. Please make sure that backup copy is the correct one and it is not corrupted.
XSLAT	The log directory '<*directoryName*>' exists. The directory might belong to another database. Check that the location specified for the logDevice attribute is correct.

Table 87. Class XSLB: RawStore - Log.Generic statement exceptions

SQLSTATE	Message Text
XSLB1	Log operation <*logOperation*> encounters error writing itself out to the log stream, this could be caused by an errant log operation or internal log buffer full due to excessively large log operation.
XSLB2	Log operation <*logOperation*> logging excessive data, it filled up the internal log buffer.
XSLB4	Cannot find truncationLWM <*value*>.
XSLB5	Illegal truncationLWM instant <*value*> for truncation point <*value*>. Legal range is from <*value*> to <*value*>.
XSLB6	Trying to log a 0 or -ve length log Record.
XSLB8	Trying to reset a scan to <*value*>, beyond its limit of <*value*>.
XSLB9	Cannot issue any more change, log factory has been stopped.

Table 88. Class XSRS: RawStore - protocol.Interface statement

SQLSTATE	Message Text
XSRS0	Cannot freeze the database after it is already frozen.
XSRS1	Cannot backup the database to <*value*>, which is not a directory.
XSRS4	Error renaming file (during backup) from <*value*> to <*value*>.
XSRS5	Error copying file (during backup) from <*path*> to <*path*>.
XSRS6	Cannot create backup directory <*directoryName*>.
XSRS7	Backup caught unexpected exception.
XSRS8	Log Device can only be set during database creation time, it cannot be changed on the fly.
XSRS9	Record <*recordName*> no longer exists
XSRSA	Cannot backup the database when unlogged operations are uncommitted. Please commit the transactions with backup blocking operations.
XSRSB	Backup cannot be performed in a transaction with uncommitted unlogged operations.
XSRSC	Cannot backup the database to <*directoryLocation*>, it is a database directory.

Table 89. Class XSTA2: XACT_TRANSACTION_ACTIVE

SQLSTATE	Message Text
XSTA2	A transaction was already active, when attempt was made to make another transaction active.

Table 90. Class XSTB: RawStore - Transactions.Basic system

SQLSTATE	Message Text
XSTB0	An exception was thrown during transaction abort.
XSTB2	Cannot log transaction changes, maybe trying to write to a read only database.
XSTB3	Cannot abort transaction because the log manager is null, probably due to an earlier error.
XSTB5	Creating database with logging disabled encountered unexpected problem.
XSTB6	Cannot substitute a transaction table with another while one is already in use.

Table 91. Class XXXXX: No SQLSTATE

SQLSTATE	Message Text
XXXXX	Normal database session close.

JDBC reference

Derby comes with a built-in JDBC driver.

That makes the JDBC API the only API for working with Derby databases. The driver is a native protocol all-Java driver (type number four of types defined by Sun).

This section provides reference information about Derby's implementation of the JDBC API and documents the way it conforms to the JDBC 3.0 and 4.0 APIs.

See the *Java DB Developer's Guide* for task-oriented instructions on working with the driver.

This JDBC driver implements the standard JDBC interface defined by Sun. When invoked from an application running in the same JVM as Derby, the JDBC driver supports connections to a Derby database in embedded mode. No network transport is required to access the database. In client/server mode, the client application dispatches JDBC requests to the JDBC server over a network; the server, in turn, which runs in the same JVM as Derby, sends requests to Derby through the embedded JDBC driver.

The Derby JDBC implementation provides access to Derby databases and supplies all the required JDBC interfaces. Unimplemented aspects of the JDBC driver return an *SQLException* with a message stating "Feature not implemented" and an *SQLState* of XJZZZ. These unimplemented parts are for features not supported by Derby.

java.sql.Driver interface

The class that loads Derby's local JDBC driver is the class *org.apache.derby.jdbc.EmbeddedDriver*. The class that loads Derby's network client driver is the class *org.apache.derby.jdbc.ClientDriver*. Listed below are some of the ways to create instances of these classes. Do not use the classes directly through the *java.sql.Driver* interface. Use the *DriverManager* class to create connections.

If your application runs on JDK 6 or higher, you do not need to do any of the following. The driver will load automatically when your application asks for its first connection.

- ```
  Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
  Class.forName("org.apache.derby.jdbc.ClientDriver");
  ```

  The recommended way to load the driver class.

  With the embedded driver, if your application shuts down Derby or calls the *DriverManager.unload* method, and you then want to reload the driver, call the *Class.forName().newInstance()* method to do so:

  ```
  Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
  ```
- ```
  new org.apache.derby.jdbc.EmbeddedDriver();
  new org.apache.derby.jdbc.ClientDriver();
  ```

  Same as using *Class.forName()*, except that it requires the class to be found when the code is compiled.
- ```
  Class c = org.apache.derby.jdbc.EmbeddedDriver.class;
  Class c = org.apache.derby.jdbc.ClientDriver.class;
  ```

  This is also the same as using *Class.forName()*, except that it requires the class to be found when the code is compiled. The pseudo-static field *class* evaluates to the class that is named.
- *Setting the system property jdbc.drivers*
  ```

To set a system property, you alter the invocation command line or the system properties within your application. It is not possible to alter system properties within an applet.

```
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
    applicationClass

java -Djdbc.drivers=org.apache.derby.jdbc.ClientDriver
    applicationClass
```

The actual driver that gets registered in the *DriverManager* to handle the *jdbc:derby:* protocol is not the class *org.apache.derby.jdbc.EmbeddedDriver* or *org.apache.derby.jdbc.ClientDriver*; that class simply detects the type of Derby driver needed and then causes the appropriate Derby driver to be loaded.

The only supported way to connect to a Derby system through the *jdbc:derby:* protocol is using the *DriverManager* to obtain a driver (*java.sql.Driver*) or connection (*java.sql.Connection*) through the *getDriver* and *getConnection* method calls.

java.sql.Driver.getPropertyInfo method

To get the *DriverPropertyInfo* object, request the JDBC driver from the driver manager:

```
java.sql.DriverManager.getDriver("jdbc:derby:").
    getPropertyInfo(URL, Prop)
```

Do not request it from *org.apache.derby.jdbc.EmbeddedDriver*, which is only an intermediary class that loads the actual driver.

This method might return a *DriverPropertyInfo* object. In a Derby system, it consists of an array of database connection URL attributes. The most useful attribute is *databaseName=nameofDatabase*, which means that the object consists of a list of booted databases in the current system.

For example, if a Derby system has the databases *toursDB* and *flightsDB* in its system directory, all the databases in the system are set to boot automatically, and a user has also connected to a database `A:/dbs/tours94`, the array returned from *getPropertyInfo* contains one object corresponding to the *databaseName* attribute. The choices field of the *DriverPropertyInfo* object will contain an array of three Strings with the values *toursDB*, *flightsDB*, and `A:/dbs/tours94`. Note that this object is returned only if the proposed connection objects do not already include a database name (in any form) or include the shutdown attribute with the value true.

For more information about *java.sql.Driver.getPropertyInfo*, see "Offering connection choices to the user" in the *Java DB Developer's Guide*.

java.sql.DriverManager.getConnection method

A Java application using the JDBC API establishes a connection to a database by obtaining a *Connection* object. The standard way to obtain a *Connection* object is to call the method *DriverManager.getConnection*, which takes a String containing a database connection URL. A JDBC database connection URL (uniform resource locator) provides a way of identifying a database.

DriverManager.getConnection can take one argument besides a database connection URL, a *Properties* object. You can use the *Properties* object to set database connection URL attributes.

You can also supply strings representing user names and passwords. When they are supplied, Derby checks whether they are valid for the current system if user

authentication is enabled. User names are passed to Derby as authorization identifiers, which are used to determine whether the user is authorized for access to the database and for determining the default schema. When the connection is established, if no user is supplied, Derby sets the default user to *APP*, which Derby uses to name the default schema. If a user is supplied, the default schema is the same as the user name.

Derby database connection URL syntax

A Derby database connection URL consists of the basic database connection URL followed by an optional subsubprotocol and optional attributes.

This section provides reference information only. For a more complete description, including examples, see "Connecting to Databases" in Chapter 1 of the *Java DB Developer's Guide*.

Syntax of database connection URLs for applications with embedded databases

For applications with embedded databases, the syntax of the database connection URL is

```
jdbc:derby: [subsubprotocol:][databasename][;attributes]*
```

- *jdbc:derby:*

  In JDBC lingo, *derby* is the *subprotocol* for connecting to a Derby database. The subprotocol is always *derby* and does not vary.
- *subsubprotocol:*

  subsubprotocol, which is not typically specified, specifies where Derby looks for a database: in a directory, in a classpath, or in a jar file. It is used only in rare instances, usually for read-only databases. *subsubprotocol* is one of the following:

  - directory
  - classpath: Databases are treated as read-only databases, and all *databaseNames* must begin with at least a slash, because you specify them "relative" to the classpath directory or archive. (You do not have to specify classpath as the subsubprotocol; it is implied.)
  - jar Databases are treated as read-only databases.

  jar: requires an additional element immediately before the *databaseName*:

  ```
  (pathToArchive)
  ```

  pathToArchive is the path to the jar or zip file that holds the database and includes the name of the jar or zip file.

  See the *Java DB Developer's Guide* for examples of database connection URLs for read-only databases.
- *databaseName*

  Specify the *databaseName* to connect to an existing database or a new one.

  You can specify the database name alone, or with a relative or absolute path. See "Standard Connections-Connecting to Databases in the File System" in Chapter 1 of the *Java DB Developer's Guide*.
- *attributes*

  Specify 0 or more database connection URL attributes as detailed in Attributes of the Derby database connection URL.

Additional SQL syntax

Derby also supports the following SQL standard syntax to obtain a reference to the current connection in a database-side JDBC routine:

```
jdbc:default:connection
```

Attributes of the Derby database connection URL

You can supply an optional list of attributes to a database connection URL. Derby translates these attributes into properties, so you can also set attributes in a *Properties* object passed to *DriverManager.getConnection*. (You cannot set those attributes as system properties, only in an object passed to the *DriverManager.getConnection* method.)

These attributes are specific to Derby and are listed in Setting attributes for the database connection URL.

Attribute name/value pairs are converted into properties and added to the properties provided in the connection call. If no properties are provided in the connection call, a properties set is created that contains only the properties obtained from the database connection URL.

```
import java.util.Properties;

Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB;create=true");

/* setting an attribute in a Properties object */
Properties myProps = new Properties();
myProps.put("create", "true");
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB", myProps);

/* passing user name and password */
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sampleDB", "dba", "password");
```

Note: Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored. (Derby does provide a tool for parsing the correctness of attributes. For more information, see the *Java DB Tools and Utilities Guide*.)

java.sql.Connection interface

A Derby*Connection* object is not garbage-collected until all other JDBC objects created from that connection are explicitly closed or are themselves garbage-collected. Once the connection is closed, no further JDBC requests can be made against objects created from the connection. Do not explicitly close the *Connection* object until you no longer need it for executing statements.

A session-severity or higher exception causes the connection to close and all other JDBC objects against it to be closed. System-severity exceptions cause the Derby system to shut down, which not only closes the connection but means that no new connections should be created in the current JVM.

Table 92. Implementation Notes on Connection Methods

Returns	Signature	Implementation Notes
Prepared	*prepareStatement(String sql, int [] columnIndexes)*	Every column index in the array must correlate to an auto-increment column

Returns	Signature	Implementation Notes
		within the target table of the INSERT. Supported in embedded mode only.
Prepared	*prepareStatement(String sql, String [] columnNames)*	Every column name in the array must designate an auto-increment column within the target table of the INSERT. Supported in embedded mode only.

See Autogenerated keys for details on the use of the two forms of the *Connection.prepareStatement* method shown in this table.

java.sql.Connection.setTransactionIsolation method

java.sql.Connection.TRANSACTION_SERIALIZABLE, *java.sql.Connection.TRANSACTION_REPEATABLE_READ*, *java.sql.Connection.TRANSACTION_READ_COMMITTED, and java.sql.Connection.TRANSACTION_READ_UNCOMMITTED* transaction isolations are available from a Derby database.

TRANSACTION_READ_COMMITTED is the default isolation level.

Changing the current isolation for the connection with *setTransactionIsolation* commits the current transaction and begins a new transaction. For more details about transaction isolation, see "Locking, concurrency, and isolation" in the *Java DB Developer's Guide*.

java.sql.Connection.setReadOnly method

java.sql.Connection.setReadOnly is supported.

See the section "Differences using the Connection.setReadOnly method" in the *Java DB Server and Administration Guide* for more information.

java.sql.Connection.isReadOnly method

If you connect to a read-only database, the appropriate *isReadOnly DatabaseMetaData* value is returned. For example, *Connections* set to read-only using the *setReadOnly* method, *Connections* for which the user has been defined as a *readOnlyAccess* user (with one of the Derby properties), and *Connections* to databases on read-only media return true.

Connection functionality not supported

Derby does not use catalog names. In addition, the following optional methods raise "Feature not supported" exceptions:

- *createArrayOf(java.lang.String, java.lang.Object[])*
- *createNClob( )*
- *createSQLXML( )*
- *createStruct(java.lang.String, java.lang.Object[])*
- *getTypeMap( )*
- *prepareStatement(java.lang.String, int[])*
- *prepareStatement(java.lang.String, java.lang.String[])*
- *setTypeMap(java.util.Map)*

java.sql.DatabaseMetaData interface

This section discusses `java.sql.DatabaseMetaData` functionality in Derby.

The Derby implementation of the *getResultSetHoldability* method returns *ResultSet.HOLD_CURSORS_OVER_COMMIT*.

DatabaseMetaData result sets

DatabaseMetaData result sets do not close the result sets of other statements, even when auto-commit is set to true.

DatabaseMetaData result sets are closed if a user performs any other action on a JDBC object that causes an automatic *commit* to occur. If you need the *DatabaseMetaData* result sets to be accessible while executing other actions that would cause automatic commits, turn off auto-commit with *setAutoCommit(false)*.

java.sql.DatabaseMetaData.getProcedureColumns method

Derby supports Java procedures. Derby allows you to call Java procedures within SQL statements. Derby returns information about the parameters in the *getProcedureColumns* call. If the corresponding Java method is overloaded, it returns information about each signature separately. Derby returns information for all Java procedures defined by CREATE PROCEDURE.

getProcedureColumns returns a *ResultSet*. Each row describes a single parameter or return value.

Parameters to getProcedureColumns

The JDBC API defines the following parameters for this method call:

- *catalog*

  always use *null* for this parameter in Derby.
- *schemaPattern*

  Java procedures have a schema.
- *procedureNamePattern*

  a String object representing a procedure name pattern.
- *column-Name-Pattern*

  a String object representing the name pattern of the parameter names or return value names. Java procedures have parameter names matching those defined in the CREATE PROCEDURE statement. Use "%" to find all parameter names.

Columns in the ResultSet returned by *getProcedureColumns*

Columns in the *ResultSet* returned by *getProcedureColumns* are as described by the API. Further details for some specific columns:

- PROCEDURE_CAT

  always "null" in Derby
- PROCEDURE_SCHEM

  schema for a Java procedure
- PROCEDURE_NAME

  the name of the procedure
- COLUMN_NAME

  the name of the parameter (see column-Name-Pattern)

- COLUMN_TYPE

  short indicating what the row describes. Always is *DatabaseMetaData.procedureColumnIn* for method parameters, unless the parameter is an array. If so, it is *DatabaseMetaData.procedureColumnInOut*. It always returns *DatabaseMetaData.procedureColumnReturn* for return values.
- TYPE_NAME

  Derby-specific name for the type.
- NULLABLE

  always returns *DatabaseMetaData.procedureNoNulls* for primitive parameters and *DatabaseMetaData.procedureNullable* for object parameters
- REMARKS

  a String describing the java type of the method parameter
- COLUMN_DEF

  a String describing the default value for the column (may be null)
- SQL_DATA_TYPE

  reserved by JDBC spec for future use
- SQL_DATETIME_SUB

  reserved by JDBC spec for future use
- CHAR_OCTET_LENGTH

  the maximum length of binary and character based columns (or any other datatype the returned value is a NULL)
- ORDINAL_POSITION

  the ordinal position, starting from 1, for the input and output parameters for a procedure.
- IS_NULLABLE

  a String describing the parameter's nullability (YES means parameter can include NULLs, NO means it can't)
- SPECIFIC_NAME

  the name which uniquely identifies this procedure within its schema
- METHOD_ID

  a Derby-specific column.
- PARAMETER_ID

  a Derby-specific column.

java.sql.DatabaseMetaData.getBestRowIdentifier method

The *java.sql.DatabaseMetaData.getBestRowIdentifier* method looks for identifiers in a specific order. This order might not return a unique row.

The *java.sql.DatabaseMetaData.getBestRowIdentifier* method looks for identifiers in the following order:

- A primary key on the table
- A unique constraint or unique index on the table
- All of the columns in the table

Note: If the *java.sql.DatabaseMetaData.getBestRowIdentifier* method does not find a primary key, unique constraint, or unique index, the method must look for identifiers in all of the columns in the table. When the method looks for identifiers this way, the method will always find a set of columns that identify a row. However, a unique row might not be identified if there are duplicate rows in the table.

java.sql.Statement interface

Derby does not implement the *setEscapeProcessing* method of *java.sql.Statement*. In addition, the *cancel* method raises a "Feature not supported" exception.

Table 93. Implementation Notes on Statement Methods

Returns	Signature	Implementation Notes
ResultSet	*getGeneratedKeys()*	If the user has indicated that auto-generated keys should be made available, this method returns the same results as a call to the IDENTITY_VAL_LOCAL function. Otherwise this method returns null.
boolean	*execute(String sql, int [] columnIndexes)*	Every column index in the array must correlate to an auto-increment column within the target table of the INSERT. Supported in embedded mode only.
boolean	*execute(String sql, String [] columnNames)*	Every column name in the array must designate an auto-increment column within the target table of the INSERT. Supported in embedded mode only.
int	*executeUpdate(String sql, int [] columnIndexes)*	Every column index in the array must correlate to an auto-increment column within the target table of the INSERT. Supported in embedded mode only.
int	*executeUpdate(String sql, String [] columnNames)*	Every column name in the array must designate an auto-increment column within the target table of the INSERT. Supported in embedded mode only.

ResultSet objects

An error that occurs when a SELECT statement is first executed prevents a *ResultSet* object from being opened on it. The same error does not close the *ResultSet* if it occurs after the *ResultSet* has been opened.

For example, a divide-by-zero error that happens while the *executeQuery* method is called on a *java.sql.Statement* or *java.sql.PreparedStatement* throws an exception and returns no result set at all, while if the same error happens while the *next* method is called on a *ResultSet* object, it does not cause the result set to be closed.

Errors can happen when a *ResultSet* is first being created if the system partially executes the query before the first row is fetched. This can happen on any query that uses more than one table and on queries that use aggregates, GROUP BY, ORDER BY, DISTINCT, INTERSECT, EXCEPT, or UNION.

Closing a *Statement* causes all open *ResultSet* objects on that statement to be closed as well.

The cursor name for the cursor of a *ResultSet* can be set before the statement is executed. However, once it is executed, the cursor name cannot be altered.

Autogenerated keys

JDBC's auto-generated keys feature provides a way to retrieve values from columns that are part of an index or have a default value assigned. Derby supports the auto-increment feature, which allows users to create columns in tables for which the database system automatically assigns increasing integer values. Users can call the method *Statement.getGeneratedKeys* to retrieve the value of such a column. This method returns a *ResultSet* object with a column for the automatically generated key. Calling *ResultSet.getMetaData* on the *ResultSet* object returned by *getGeneratedKeys* produces a *ResultSetMetaData* object that is similar to that returned by IDENTITY_VAL_LOCAL.

Users can indicate that auto-generated columns should be made available for retrieval by passing one of the following values as a second argument to the *Connection.prepareStatement*, *Statement.execute*, or *Statement.executeUpdate* methods:

- A constant indicating that auto-generated keys should be made available. The specific constant to use is `Statement.RETURN_GENERATED_KEYS`.
- An array of the names of the columns in the inserted row that should be made available. If any column name in the array does *not* designate an auto-increment column, Derby will throw an error with the Derby embedded driver. With the client driver, the one element column name is ignored currently and the value returned corresponds to the identity column. To ensure compatibility with future changes an application should ensure the column described is the identity column. If the column name corresponds to another column or a non-existent column then future changes may result in a value for a different column being returned or an exception being thrown.
- An array of the positions of the columns in the inserted row that should be made available. If any column position in the array does *not* correlate to an auto-increment column, Derby will throw an error with the Derby embedded driver. With the client driver, the one element position array is ignored currently and the value returned corresponds to the identity column. To ensure compatibility with future changes an application should ensure the column described is the identity column. If the position corresponds to another column or a non-existent column then future changes may result in a value for a different column being returned or an exception being thrown.

Example

Assume that we have a table TABLE1 defined as follows:

```
CREATE TABLE TABLE1 (C11 int, C12 int GENERATED ALWAYS AS IDENTITY)
```

The following three code fragments will all do the same thing: that is, they will create a *ResultSet* that contains the value of `C12` that is inserted into TABLE1.

Code fragment 1:

```
Statement stmt = conn.createStatement();
stmt.execute(
    "INSERT INTO TABLE1 (C11) VALUES (1)",
    Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
```

Code fragment 2:

```
Statement stmt = conn.createStatement();
String [] colNames = new String [] { "C12" };
stmt.execute(
    "INSERT INTO TABLE1 (C11) VALUES (1)",
    colNames);
ResultSet rs = stmt.getGeneratedKeys();
```

Code fragment 3:

```
Statement stmt = conn.createStatement();
int [] colIndexes = new int [] { 2 };
stmt.execute(
    "INSERT INTO TABLE1 (C11) VALUES (1)",
    colIndexes);
ResultSet rs = stmt.getGeneratedKeys();
```

If there is no indication that auto-generated columns should be made available for retrieval, a call to *Statement.getGeneratedKeys* will return a null *ResultSet*.

java.sql.CallableStatement interface

Derby supports all methods of *CallableStatement* except *setBlob*, *getBlob*, *setClob*, and *getClob*.

CallableStatements and OUT Parameters

Derby supports OUT parameters and CALL statements that return values, as in the following example:

```
CallableStatement cs = conn.prepareCall(
   "? = CALL getDriverType(cast (? as INT))"
cs.registerOutParameter(1, Types.INTEGER);
cs.setInt(2, 35);
cs.executeUpdate();
```

Note: Using a CALL statement with a procedure that returns a value is only supported with the *?* = syntax.

Register the output type of the parameter before executing the call.

CallableStatements and INOUT Parameters

INOUT parameters map to an *array* of the parameter type in Java. (The method must take an array as its parameter.) This conforms to the recommendations of the SQL standard.

Given the following example:

```
CallableStatement call = conn.prepareCall(
    "{CALL doubleMyInt(?)}");
// for inout parameters, it is good practice to
// register the outparameter before setting the input value
call.registerOutParameter(1, Types.INTEGER);
call.setInt(1,10);
call.execute();
int retval = call.getInt(1);
```

The method *doubleIt* should take a one-dimensional array of `ints`. Here is sample source code for that method:

```
public static void doubleMyInt(int[] i) {
    i[0] *=2;
    /* Derby  returns the first element of the array.*/
```

```
}
```

Note: The return value is *not* wrapped in an array even though the parameter to the method is.

Table 94. INOUT Parameter Type Correspondence

JDBC Type	Array Type for Method Parameter	Value and Return Type
BIGINT	long[]	long
BINARY	byte[][]	byte[]
BIT	boolean[]	boolean
DATE	*java.sql.Date[]*	*java.sql.Date*
DOUBLE	double[]	double
FLOAT	double[]	double
INTEGER	int[]	int
LONGVARBINARY	byte[][]	byte[]
REAL	float[]	float
SMALLINT	short[]	short
TIME	*java.sql.Time[]*	*java.sql.Time*
TIMESTAMP	*java.sql.Timestamp[]*	*java.sql.Timestamp*
VARBINARY	byte[][]	byte[]
OTHER	*yourType[]*	*yourType*
JAVA_OBJECT (only valid in Java2/JDBC 2.0 environments)	*yourType[]*	*yourType*

Register the output type of the parameter before executing the call. For INOUT parameters, it is good practice to register the output parameter before setting its input value.

java.sql.PreparedStatement interface

Derby provides all the required JDBC type conversions and additionally allows use of the individual *setXXX* methods for each type as if a *setObject(Value, JDBCTypeCode)* invocation were made.

This means that *setString* can be used for any built-in target type.

The *setCursorName* method can be used on a *PreparedStatement* prior to an execute request to control the cursor name used when the cursor is created.

Prepared statements and streaming columns

setXXXStream requests stream data between the application and the database.

JDBC allows an IN parameter to be set to a Java input stream for passing in large amounts of data in smaller chunks. When the statement is run, the JDBC driver makes repeated calls to this input stream. Derby supports the following JDBC stream methods for *PreparedStatement* objects:

- *setBinaryStream*

Use for streams that contain uninterpreted bytes
- *setAsciiStream*

  Use for streams that contain ASCII characters
- *setCharacterStream*

  Use for streams that contain characters

Note: Derby does not support the *setNCharacterStream* method or the deprecated *setUnicodeStream* method.

JDBC 3.0 requires that you specify the length of the stream, and Derby enforces this requirement if your application runs on JDK 5 or earlier. If your application runs on JDK 6, Derby exposes a JDBC 4.0 implementation, which lets you use the streaming interfaces without having to specify the stream length. The stream object passed to *setBinaryStream* and *setAsciiStream* can be either a standard Java stream object or the user's own subclass that implements the standard *java.io.InputStream* interface. The object passed to *setCharacterStream* must be a subclass of the abstract *java.io.Reader* class.

According to the JDBC standard, streams can be stored only in columns with the data types shown in the following table.

Table 95. Streamable JDBC Data Types

Column Data Type	Corresponding Java Type	AsciiStream	CharacterStream	BinaryStream
CLOB	java.sql.Clob	x	x	
CHAR		x	x	
VARCHAR		x	x	
LONGVARCHAR		X	X	
BINARY		x	x	x
BLOB	java.sql.Blob	x	x	x
VARBINARY		x	x	x
LONGVARBINARY		x	x	X

Note:
- A large X indicates the preferred target data type for the type of stream. See Mapping of java.sql.Types to SQL Types.
- For applications using the client driver, if the stream is stored in a column of a type other than LONG VARCHAR or LONG VARCHAR FOR BIT DATA, the entire stream must be able to fit into memory at one time. Streams stored in LONG VARCHAR and LONG VARCHAR FOR BIT DATA columns do not have this limitation.
- Streams cannot be stored in columns of the other built-in data types or columns of user-defined data types.

Example

The following code fragment shows how a user can store a streamed, ASCII-encoded *java.io.File* in a LONG VARCHAR column:

```
Statement s = conn.createStatement();
s.executeUpdate("CREATE TABLE atable (a INT, b LONG VARCHAR)");
conn.commit();
```

```
    java.io.File file = new java.io.File("derby.txt");
    int fileLength = (int) file.length();

    // create an input stream
    java.io.InputStream fin = new java.io.FileInputStream(file);
    PreparedStatement ps = conn.prepareStatement(
        "INSERT INTO atable VALUES (?, ?)");
    ps.setInt(1, 1);

    // set the value of the input parameter to the input stream
    ps.setAsciiStream(2, fin, fileLength);
    ps.execute();
    conn.commit();
```

java.sql.ResultSet interface

A positioned update or delete issued against a cursor being accessed through a *ResultSet* object modifies or deletes the current row of the *ResultSet* object.

Some intermediate protocols might pre-fetch rows. This causes positioned updates and deletes to operate against the row the underlying cursor is on, and not the current row of the *ResultSet*.

JDBC does not define the sort of rounding to use for *ResultSet.getBigDecimal*. Derby uses *java.math.BigDecimal.ROUND_HALF_DOWN*.

Table 96. Implementation Notes on *ResultSet* Methods

Returns	Signature	Implementation Notes
void	*deleteRow()*	After the row is deleted, the *ResultSet* object will be positioned before the next row. Before issuing any methods other than close on the *ResultSet* object, the program will need to reposition the *ResultSet* object.
int	*getConcurrency()*	If the Statement object has *CONCUR_READ_ONLY* concurrency, then this method will return *ResultSet.CONCUR_READ_ONLY*. But if the Statement object has *CONCUR_UPDATABLE* concurrency, then the return value will depend on whether the underlying language *ResultSet* is updatable or not. If the language *ResultSet* is updatable, then *getConcurrency()* will return *ResultSet.CONCUR_UPDATABLE*. If the language *ResultSet* is not updatable, then *getConcurrency()* will return *ResultSet.CONCUR_READ_ONLY*.
boolean	*rowDeleted()*	For forward-only result sets this method always returns *false*, for scrollable result sets it returns *true* if the row has been deleted, via result set or positioned delete.
boolean	*rowInserted()*	Always returns *false*.
boolean	*rowUpdated()*	For forward-only result sets this method always returns *false*, for scrollable result sets it returns *true* if the row has been

Returns	Signature	Implementation Notes
		updated, via result set or positioned update.
void	*updateRow()*	After the row is updated, the *ResultSet* object will be positioned before the next row. Before issuing any methods other than close on the *ResultSet* object, the program will need to reposition the *ResultSet* object.

ResultSets and streaming columns

If the underlying object is itself an *OutputStream* class, *getBinaryStream* returns the object directly.

To get a field from the *ResultSet* using streaming columns, you can use the *getXXXStream* methods if the type supports it. See Streamable JDBC Data Types for a list of types that support the various streams. (See also Mapping of java.sql.Types to SQL Types.)

You can retrieve data from one of the supported data type columns as a stream, whether or not it was stored as a stream.

The following code fragment shows how a user can retrieve a LONG VARCHAR column as a stream:

```
    // retrieve data as a stream
    ResultSet rs = s.executeQuery("SELECT b FROM atable");
    while (rs.next()) {
        // use a java.io.Reader to get the data
        java.io.Reader ip = rs.getCharacterStream(1);

        // process the stream--this is just a generic way to
        // print the data
        char[] buff = new char[128];
        int size;
        while ((size = ip.read(buff)) != -1) {
            String chunk = new String(buff, 0, size);
            System.out.print(chunk);
        }
    }
    rs.close();
    s.close();
    conn.commit();
```

java.sql.ResultSetMetaData interface

Derby does not track the source or updatability of columns in *ResultSets*, and so always returns the following constants for the following methods:

Method Name	Value
isDefinitelyWritable	false
isReadOnly	false
isWritable	false

java.sql.SQLException class

Derby supplies values for the *getMessage()*, *getSQLState()*, and *getErrorCode()* calls of *SQLExceptions*. In addition, Derby sometimes returns multiple *SQLExceptions* using the *nextException* chain. The first exception is always the most severe exception, with SQL-92 Standard exceptions preceding those that are specific to Derby. For information on processing *SQLExceptions*, see "Working with Derby SQLExceptions in an application" in the *Java DB Developer's Guide*.

java.sql.SQLWarning class

Derby can generate a warning in certain circumstances. A warning is generated if, for example, you try to connect to a database with the *create* attribute set to *true* if the database already exists. Aggregates like sum() also raise a warning if NULL values are encountered during the evaluation.

All other informational messages are written to the Derby system's *derby.log* file.

java.sql.Savepoint interface

The *Savepoint* interface contains methods to set, release, or roll back a transaction to designated savepoints. Once a savepoint has been set, the transaction can be rolled back to that savepoint without affecting preceding work. Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction.

Derby does not support savepoints within a trigger.

Derby does not release locks as part of the rollback to savepoint.

For more information on using savepoints, see the *Java DB Developer's Guide*.

Mapping of java.sql.Types to SQL types

In Derby, the java.sql.Types are mapped to SQL data types

The following table shows the mapping of *java.sql.Types* to SQL types.

Table 97. Mapping of java.sql.Types to SQL Types

java.sql.Types	**SQL Types**
BIGINT	BIGINT
BINARY	CHAR FOR BIT DATA
BIT[1]	CHAR FOR BIT DATA
BLOB	BLOB (JDBC 2.0 and up)
CHAR	CHAR
CLOB	CLOB (JDBC 2.0 and up)
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	DOUBLE PRECISION
FLOAT	DOUBLE PRECISION[2]
INTEGER	INTEGER
LONGVARBINARY	LONG VARCHAR FOR BIT DATA

java.sql.Types	SQL Types
LONGVARCHAR	LONG VARCHAR
NULL	Not a data type; always a value of a particular type
NUMERIC	DECIMAL
REAL	REAL
SMALLINT	SMALLINT
SQLXML[3]	XML
TIME	TIME
TIMESTAMP	TIMESTAMP
VARBINARY	VARCHAR FOR BIT DATA
VARCHAR	VARCHAR

Notes:

1. BIT is only valid in JDBC 2.0 and earlier environments.
2. Values can be passed in using the FLOAT type code; however, these are stored as DOUBLE PRECISION values, and so always have the type code DOUBLE when retrieved.
3. SQLXML is only valid in JDBC 4.0 and later environments. SQLXML corresponds to the SQL type XML in Derby. However, Derby does not recognize the java.sql.Types.SQLXML data type and does not support any JDBC-side operations for the XML data type. Support for XML and the related operators is implemented only at the SQL layer. See XML data types for more.

Mapping of java.sql.Blob and java.sql.Clob interfaces

In the JDBC API, *java.sql.Blob* is the mapping for the SQL BLOB (binary large object) type; *java.sql.Clob* is the mapping for the SQL CLOB (character large object) type. BLOB and CLOB objects are collectively referred to as LOBs (large objects).

The Derby implementation of the *java.sql.Blob* and *java.sql.Clob* interfaces is LOCATOR-based, meaning that the implementation provides a logical pointer to a LOB rather than a complete copy of the object. Also, Derby does not materialize a LOB when you use the BLOB or CLOB data type. You can, however, call methods on a *java.sql.Blob* and *java.sql.Clob* object to materialize it (that is, to retrieve the entire object or parts of it).

Derby implements all of the methods for these interfaces except for the *setBlob*, *getBlob*, *setClob*, and *getClob* methods of the *CallableStatement* interface.

To use the *java.sql.Blob* and *java.sql.Clob* features:

- Use the SQL BLOB type for columns which hold very large binary values.
- Use the SQL CLOB type for columns which hold very large string values.
- Use the *getBlob* and *getClob* methods of the *java.sql.ResultSet* interface to retrieve a LOB using its locator. You can then materialize all or part of the LOB by calling *Blob* and *Clob* methods. Alternatively, you can call the *ResultSet.getBytes* method to materialize a BLOB, and you can call the *ResultSet.getString* method to materialize a CLOB.

Casting between strings and BLOBs is not recommended because casting is platform- and database-dependent.

As with other character datatypes, Derby treats CLOBs as unicode strings and writes them to disk using UTF8 encoding. With a Java database like Derby, you do not need to worry about character sets and codepages.

Restrictions on BLOB and CLOB objects (LOB-types)

- LOB-types cannot be compared for equality (=) and non-equality (!=, <>).
- LOB-typed values are not orderable, so <, <=, >, >= tests are not supported.
- LOB-types cannot be used in indices or as primary key columns.
- DISTINCT, GROUP BY, and ORDER BY clauses are also prohibited on LOB-types.
- LOB-types cannot be involved in implicit casting as other base-types.

Recommendation: Because the lifespan of a *java.sql.Blob* or *java.sql.Clob* ends when the transaction commits, turn off auto-commit with the *java.sql.Blob* or *java.sql.Clob* features.

Table 98. Implementation Notes on *java.sql.Blob* Methods

Returns	Signature	Implementation Notes
byte[]	*getBytes(long pos, int length)*	Exceptions are raised if *pos* < 1, if *pos* is larger than the length of the , or if *length* <= 0.
long	*position(byte[] pattern, long start)*	Exceptions are raised if *pattern* == null, if *start* < 1, or if *pattern* is an array of length 0.
long	*position(Blob pattern, long start)*	Exceptions are raised if *pattern* == null, if *start* < 1, if *pattern* has length 0, or if an exception is thrown when trying to read the first byte of *pattern*.

Table 99. Implementation Notes on *java.sql.Clob* Methods

Returns	Signature	Implementation Notes
String	*getSubString(long pos, int length)*	Exceptions are raised if *pos* < 1, if *pos* is larger than the length of the *Clob*, or if length <= 0.
long	*position(Clob searchstr, long start)*	Exceptions are raised if *searchStr* == null or *start* < 1, if *searchStr* has length 0, or if an exception is thrown when trying to read the first char of *searchStr.*
long	*position(String searchstr, long start)*	Exceptions are raised if *searchStr* == null or start < 1, or if the pattern is an empty string.

Notes on mapping of java.sql.Blob and java.sql.Clob interfaces

The usual Derby locking mechanisms (shared locks) prevent other transactions from updating or deleting the database item to which the *java.sql.Blob* or *java.sql.Clob* object is a pointer. However, in some cases, Derby's instantaneous lock mechanisms could allow a period of time in which the column underlying the *java.sql.Blob* or *java.sql.Clob* is unprotected. A subsequent call to *getBlob/getClob*, or to a *java.sql.Blob/java.sql.Clob*method, could cause undefined behavior.

Furthermore, there is nothing to prevent the transaction that holds the *java.sql.Blob/java.sql.Clob* (as opposed to another transaction) from updating the underlying row. (The same problem exists with the *getXXXStream* methods.) Program applications to prevent updates to the underlying object while a

java.sql.Blob/*java.sql.Clob* is open on it; failing to do this could result in undefined behavior.

Do not call more than one of the *ResultSet getXXX* methods on the same column if one of the methods is one of the following:

- *getBlob*
- *getClob*
- *getAsciiStream*
- *getBinaryStream*
- *getCharacterStream*

These methods share the same underlying stream; calling more than one of these methods on the same column could result in undefined behavior. For example:

```
    ResultSet rs = s.executeQuery("SELECT text FROM CLOBS WHERE i = 1");
    while (rs.next()) {
        aclob = rs.getClob(1);
        ip = rs.getAsciiStream(1);
    }
```

The streams that handle long-columns are not thread safe. This means that if a user chooses to open multiple threads and access the stream from each thread, the resulting behavior is undefined.

Clobs are not locale-sensitive.

JDBC Package for Connected Device Configuration/Foundation Profile (JSR 169)

Derby supports the JDBC API defined for the Connected Device Configuration/Foundation Profile, also known as JSR 169. The features supported are a subset of the JDBC 3.0 specification. Support for JSR 169 is limited to the embedded driver. Derby does not support using the Network Server under JSR 169.

To obtain a connection using JSR 169, use the `org.apache.derby.jdbc.EmbeddedSimpleDataSource` class. This class is identical in implementation to the `org.apache.derby.jdbc.EmbeddedDataSource` class. See the *Java DB Developer's Guide* for information on using the properties of the `org.apache.derby.jdbc.EmbeddedDataSource` class.

JSR 169 and its Derby implementation have the following limitations:

- Applications must get and set `DECIMAL` values using alternate JDBC `getXXX` and `setXXX` methods, such as `getString()` and `setString()`. Any alternate method that works against a DECIMAL type with JDBC 3.0 will work in JSR 169.
- The XML data type is not supported, but an application can retrieve, update, query, or otherwise access an XML data value if it has classes for a JAXP parser and for Xalan in the classpath. Derby issues an error if either the parser or Xalan is not found. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

JSR 169 and its Derby implementation do not support the following:

- Java functions and procedures that use server-side JDBC, that is, routines declared with `CONTAINS SQL`, `READS SQL DATA`, or `MODIFIES SQL DATA` clauses
- The `DriverManager` interface (this means that you cannot use the `DriverManager.getConnection` method to obtain a connection but must use the `org.apache.derby.jdbc.EmbeddedSimpleDataSource` class instead)

- The standard URL used to obtain a connection, `jdbc:default:connection` (a runtime error may occur if the routine tries to obtain a connection using `jdbc:default:connection`)
- Diagnostic tables
- Triggers
- Encrypted databases
- Non-blocking I/O
- Java EE resource manager support, including distributed transactions
- Principal-based security
- LDAP-based authentication
- SSL/TLS encryption
- Replication

JDBC 4.0-only features

JDBC 4.0 adds some functionality to the core API. This section documents the features supported by Derby.

Note: These features are present only in a JDK 6 or higher environment.

These features are:

- **DataSources.** To support the JDBC 4.0 ease of development, Derby introduces new implementations of `javax.sql.DataSource`. See javax.sql.DataSource interface: JDBC 4.0 features.
- **Autoloading of JDBC drivers.** In earlier versions of JDBC, applications had to manually register drivers before requesting Connections. With JDBC 4.0, applications no longer need to issue a Class.forName() on the driver name; instead, the DriverManager will find an appropriate JDBC driver when the application requests a Connection.
- **SQLExceptions.** JDBC 4.0 introduces refined subclasses of SQLException. See Refined subclasses of SQLException.
- **Wrappers.** JDBC 4.0 introduces the concept of wrapped JDBC objects. This is a formal mechanism by which application servers can look for vendor-specific extensions inside standard JDBC objects like Connections, Statements, and ResultSets. For Derby, this is a vacuous exercise because Derby does not expose any of these extensions.
- **Statement events.** With JDBC 4.0, Connection pools can listen for Statement closing and Statement error events. New methods were added to *javax.sql.PooledConnection*: *addStatementEventListener* and *removeStatementEventListener*.
- **Streaming APIs.** JDBC 4.0 adds new overloads of the streaming methods in CallableStatement, PreparedStatement, and ResultSet. These are the *setXXX* and *updateXXX* methods which take *java.io.InputStream* and *java.io.Reader* arguments. The new overloads allow you to omit the length arguments or to specify *long* lengths.
- **New methods.** New methods were added to the following interfaces: `javax.sql.Connection`, `javax.sql.DatabaseMetaData`, and `javax.sql.Statement`. See java.sql.Connection interface: JDBC 4.0 features, java.sql.DatabaseMetaData interface: JDBC 4.0 features, java.sql.Statement interface: JDBC 4.0 features.

Refined subclasses of SQLException

If your application runs on JDK 6 or higher, exceptions raised by Derby will generally be one of the refined subclasses of SQLException, introduced by JDBC 4.0. These refined exceptions are raised under the conditions described by their respective javadoc.

- **java.sql.SQLClientInfoException**
- **java.sql.SQLDataException**
- **java.sql.SQLFeatureNotSupportedException**
- **java.sql.SQLIntegrityConstraintViolationException**
- **java.sql.SQLInvalidAuthorizationSpecException**
- **java.sql.SQLSyntaxErrorException**
- **java.sql.SQLTransactionRollbackException**
- **java.sql.SQLTransientConnectionException**

java.sql.Connection interface: JDBC 4.0 features

JDBC 4.0 adds new capabilities to Connections:

- **LOB creation** - New methods, *createBlob()* and *createClob()* let you create empty Blobs and Clobs, which you can then fill up before stuffing into a column.
- **Validity tracking** - The *isValid* method tells you whether your Connection is still alive.

java.sql.DatabaseMetaData interface: JDBC 4.0 features

Derby implements all of the new metadata methods added by JDBC 4.0.

- **Capability reports** - JDBC 4.0 adds new methods for querying the capabilities of a database. These include *autoCommitFailureClosesAllResultSets*, *providesQueryObjectGenerator*, *getClientInfoProperties*, and *supportsStoredFunctionsUsingCallSyntax*.
- **Column metadata -** The *getColumns* method reports *IS_AUTOINCREMENT* = YES if a column is generated.
- **Function metadata** - JDBC 4.0 adds new methods for inspecting the arguments and return types of functions, including user-defined functions. These new methods are *getFunctions* and *getFunctionColumns*. These methods behave similarly to *getProcedures* and *getProcedureColumns*.
- **Procedure metadata** - The *getProcedureColumns* method reports additional information about procedure arguments. For more information, see the javadoc for this method. The new columns in the ResultSet returned by *getProcedureColumns* are: *COLUMN_DEF*, *SQL_DATA_TYPE*, *SQL_DATETIME_SUB*, *CHAR_OCTET_LENGTH*, *ORDINAL_POSITION*, *IS_NULLABLE*, and *SPECIFIC_NAME*.
- **Schema metadata** - JDBC 4.0 adds a new *getSchemas* overload, which lets you look up schemas based on a name pattern.

java.sql.Statement interface: JDBC 4.0 features

Derby's Statements implement the following new metadata methods added by JDBC 4.0.

- **Pooling support** - JDBC 4.0 adds new methods to help application servers manage pooled Statements: *isPoolable* and *setPoolable*.
- **Validity tracking** - JDBC 4.0 lets you track the validity of a Statement through the new *isClosed* method.

javax.sql.DataSource interface: JDBC 4.0 features

Derby has added new JDBC 4.0-specific DataSources. Use these DataSources if your application runs on JDK 6 or higher.

- **org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40**
- **org.apache.derby.jdbc.EmbeddedDataSource40**
- **org.apache.derby.jdbc.EmbeddedXADataSource40**
- **org.apache.derby.jdbc.ClientConnectionPoolDataSource40**

- **org.apache.derby.jdbc.ClientDataSource40**
- **org.apache.derby.jdbc.ClientXADataSource40**

java.sql.SQLXML interface

In JDBC 4.0, java.sql.SQLXML is the mapping for the SQL XML data type. However, Derby defines the XML data type and operators only in the SQL layer. There is no JDBC-side support for the XML data type and operators

You cannot instantiate a java.sql.SQLXML object in Derby, or bind directly into an XML value or retrieve an XML value directly from a result set. You must bind and retrieve the XML data as Java strings or character streams by explicitly specifying the XML operators, XMLPARSE and XMLSERIALIZE, as part of your SQL queries.

Additionally, Derby does not provide JDBC metatadata support for the XML data type.

JDBC escape syntax

JDBC provides a way of smoothing out some of the differences in the way different DBMS vendors implement SQL. This is called escape syntax. Escape syntax signals that the JDBC driver, which is provided by a particular vendor, scans for any escape syntax and converts it into the code that the particular database understands. This makes escape syntax DBMS-independent.

A JDBC escape clause begins and ends with curly braces. A keyword always follows the opening curly brace:

```
{keyword }
```

Derby supports the following JDBC escape keywords, which are case-insensitive:

- *JDBC escape keyword for call statements*

  The escape keyword for use in *CallableStatements*.
- *JDBC escape syntax*

  The escape keyword for date formats.
- *JDBC escape syntax for LIKE clauses*

  The keyword for specifying escape characters for LIKE clauses.
- *JDBC escape syntax for fn keyword*

  The escape keyword for scalar functions.
- *JDBC escape syntax for outer joins*

  The escape keyword for outer joins.
- *JDBC escape syntax for time formats*

  The escape keyword for time formats.
- *JDBC escape syntax for timestamp formats*

  The escape keyword for timestamp formats.

Other JDBC escape keywords are not supported.
Note: Derby returns the SQL unchanged in the *Connection.nativeSQL* call, since the escape syntax is native to SQL. In addition, it is unnecessary to call *Statement.setEscapeProcessing* for this reason.

JDBC escape keyword for call statements

This syntax is supported for a *java.sql.Statement* and a *java.sql.PreparedStatement* in addition to a *CallableStatement*.

Syntax

```
{call statement }
```

```
-- Call a Java procedure
{ call TOURS.BOOK_TOUR(?, ?) }
```

JDBC escape syntax

Derby interprets the JDBC escape syntax for date as equivalent to the SQL syntax for dates.

Syntax

```
{d 'yyyy-mm-dd'}
```

Equivalent to

```
DATE('yyyy-mm-dd')
```

```
VALUES {d '1999-01-09'}
```

JDBC escape syntax for LIKE clauses

The percent sign % and underscore _ are metacharacters within SQL LIKE clauses. JDBC provides syntax to force these characters to be interpreted literally. The JDBC clause immediately following a LIKE expression allows you to specify an escape character:

Syntax

```
WHERE CharacterExpression [ NOT ]
 LIKE
    CharacterExpressionWithWildCard
    { ESCAPE 'escapeCharacter' }
```

```
-- find all rows in which a begins with the character "%"
SELECT a FROM tabA WHERE a LIKE '$%%' {escape '$'}
-- find all rows in which a ends with the character "_"
SELECT a FROM tabA WHERE a LIKE '%=_' {escape '='}
```

Note: ? is not permitted as an escape character if the LIKE pattern is also a dynamic parameter (?).

In some languages, a single character consists of more than one collation unit (a 16-bit character). The *escapeCharacter* used in the escape clause must be a single collation unit in order to work properly.

You can also use the escape character sequence for LIKE without using JDBC's curly braces; see Boolean expressions.

JDBC escape syntax for fn keyword

You can specify functions in JDBC escape syntax by using the `fn` keyword.

Syntax

```
{fn functionCall}
```

where *functionCall* is the name of one of the scalar functions listed below. The functions are of the following types:

- Numeric functions
- String functions
- Date and time functions
- System function

Numeric functions

abs

Returns the absolute value of a number.

```
abs(NumericExpression)
```

The JDBC escape syntax `{fn abs(NumericExpression)}` is equivalent to the built-in syntax `ABS(NumericExpression)`. For more information, see ABS or ABSVAL function.

acos

Returns the arc cosine of a specified number.

```
acos(number)
```

The JDBC escape syntax `{fn acos(number)}` is equivalent to the built-in syntax `ACOS(number)`. For more information, see ACOS function.

asin

Returns the arc sine of a specified number.

```
asin(number)
```

The JDBC escape syntax `{fn asin(number)}` is equivalent to the built-in syntax `ASIN(number)`. For more information, see ASIN function.

atan

Returns the arc tangent of a specified number.

```
atan(number)
```

The JDBC escape syntax `{fn atan(number)}` is equivalent to the built-in syntax `ATAN(number)`. For more information, see ATAN function.

atan2

Returns the arc tangent in radians of `y/x`.

```
atan2(y, x)
```

The JDBC escape syntax `{fn atan2(y, x)}` is equivalent to the built-in syntax `ATAN2(y, x)`. For more information, see ATAN2 function.

ceiling

Rounds the specified number up, and returns the smallest number that is greater than or equal to the specified number.

```
ceiling(number)
```

The JDBC escape syntax `{fn ceiling(number)}` is equivalent to the built-in syntax `CEILING(number)`. For more information, see CEIL or CEILING function.

cos

Returns the cosine of a specified number.

```
cos(number)
```

The JDBC escape syntax `{fn cos(number)}` is equivalent to the built-in syntax `COS(number)`. For more information, see COS function.

cot

Returns the cotangent of a specified number.

```
cot(number)
```

The JDBC escape syntax `{fn cot(number)}` is equivalent to the built-in syntax `COT(number)`. For more information, see COT function.

degrees

Converts a specified number from radians to degrees.

```
degrees(number)
```

The JDBC escape syntax `{fn degrees(number)}` is equivalent to the built-in syntax `DEGREES(number)`. For more information, see DEGREES function.

exp

Returns e raised to the power of the specified number.

```
exp(number)
```

The JDBC escape syntax `{fn exp(number)}` is equivalent to the built-in syntax `EXP(number)`. For more information, see EXP function.

floor

Rounds the specified number down, and returns the largest number that is less than or equal to the specified number.

```
floor(number)
```

The JDBC escape syntax `{fn floor(number)}` is equivalent to the built-in syntax `FLOOR(number)`. For more information, see FLOOR function.

log

Returns the natural logarithm (base e) of the specified number.

```
log(number)
```

The JDBC escape syntax `{fn log(number)}` is equivalent to the built-in syntax `LOG(number)`. For more information, see LN or LOG function.

log10

Returns the base-10 logarithm of the specified number.

```
log10(number)
```

The JDBC escape syntax `{fn log10(number)}` is equivalent to the built-in syntax `LOG10(number)`. For more information, see LOG10 function.

mod

Returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative.

```
mod(integer_type, integer_type)
```

The JDBC escape syntax `{fn mod(integer_type, integer_type)}` is equivalent to the built-in syntax `MOD(integer_type, integer_type)`. For more information, see MOD function.

pi

Returns a value that is closer than any other value to pi.

```
pi()
```

The JDBC escape syntax `{fn pi()}` is equivalent to the built-in syntax `PI()`. For more information, see PI function.

radians

Converts a specified number from degrees to radians.

```
radians(number)
```

The JDBC escape syntax `{fn radians(number)}` is equivalent to the built-in syntax `RADIANS(number)`. For more information, see RADIANS function.

rand

Returns a random number given a seed number.

```
rand(seed)
```

The JDBC escape syntax `{fn rand(seed)}` is equivalent to the built-in syntax `RAND(seed)`. For more information, see RAND function.

sign

Returns an integer that represents the sign of a specified number (+1 if the number is positive, -1 if it is negative, 0 if it is 0).

```
sign(number)
```

The JDBC escape syntax `{fn sign(number)}` is equivalent to the built-in syntax `SIGN(number)`. For more information, see SIGN function.

sin

Returns the sine of a specified number.

```
sin(number)
```

The JDBC escape syntax `{fn sin(number)}` is equivalent to the built-in syntax `SIN(number)`. For more information, see SIN function.

sqrt

Returns the square root of a floating-point number.

```
sqrt(FloatingPointExpression)
```

The JDBC escape syntax `{fn sqrt(FloatingPointExpression)}` is equivalent to the built-in syntax `SQRT(FloatingPointExpression)`. For more information, see SQRT function.

tan

Returns the tangent of a specified number.

```
tan(number)
```

The JDBC escape syntax `{fn tan(number)}` is equivalent to the built-in syntax `TAN(number)`. For more information, see TAN function.

String functions

concat

Returns the concatenation of character strings; that is, the character string formed by appending the second string to the first string. If either string is null, the result is NULL.

```
concat(CharacterExpression, CharacterExpression)
```

The JDBC escape syntax `{fn concat(CharacterExpression, CharacterExpression)}` is equivalent to the built-in syntax

CharacterExpression || *CharacterExpression*. For more information, see Concatenation operator.

lcase

Returns a string in which all alphabetic characters in the argument have been converted to lowercase.

```
lcase(CharacterExpression)
```

The JDBC escape syntax `{fn lcase(`*CharacterExpression*`)}` is equivalent to the built-in syntax `LCASE(`*CharacterExpression*`)`. For more information, see LCASE or LOWER function.

length

Returns the number of characters in a character string expression.

```
length(CharacterExpression)
```

The JDBC escape syntax `{fn length(`*CharacterExpression*`)}` is equivalent to the built-in syntax `LENGTH(`*CharacterExpression*`)`. For more information, see LENGTH function.

locate

Returns the position in the second *CharacterExpression* of the first occurrence of the first *CharacterExpression*. Searches from the beginning of the second *CharacterExpression*, unless the *startIndex* parameter is specified.

```
locate(CharacterExpression,CharacterExpression [, startIndex] )
```

The JDBC escape syntax `{fn locate(`*CharacterExpression*`,` *CharacterExpression* `[,` *startIndex*`] )}` is equivalent to the built-in syntax `LOCATE(`*CharacterExpression*`,` *CharacterExpression* `[,` *StartPosition]* `)`. For more information, see LOCATE function.

ltrim

Removes blanks from the beginning of a character string expression.

```
ltrim(CharacterExpression)
```

The JDBC escape syntax `{fn ltrim(`*CharacterExpression*`)}` is equivalent to the built-in syntax `LTRIM(`*CharacterExpression*`)`. For more information, see LTRIM function.

rtrim

Removes blanks from the end of a character string expression.

```
rtrim(CharacterExpression)
```

The JDBC escape syntax `{fn rtrim(`*CharacterExpression*`)}` is equivalent to the built-in syntax `RTRIM(`*CharacterExpression*`)`. For more information, see RTRIM function.

substring

Forms a character string by extracting *length* characters from the *CharacterExpression* beginning at *startIndex*. The index of the first character in the *CharacterExpression* is 1.

```
substring(CharacterExpression, startIndex, length)
```

The JDBC escape syntax `{fn substring(`*CharacterExpression*`,` *startIndex*`,` *length*`)}` is equivalent to the built-in syntax `SUBSTR(`*CharacterExpression*`,` *startIndex*`,` *length*`)`. For more information, see SUBSTR function.

ucase

Returns a string in which all alphabetic characters in the argument have been converted to uppercase.

```
ucase(CharacterExpression)
```

The JDBC escape syntax `{fn ucase(CharacterExpression)}` is equivalent to the built-in syntax `UCASE(CharacterExpression)`. For more information, see UCASE or UPPER function.

Date and time functions

curdate

Returns the current date.

```
curdate()
```

The JDBC escape syntax `{fn curdate()}` is equivalent to the built-in syntax `CURRENT_DATE`. For more information, see CURRENT_DATE function.

curtime

Returns the current time.

```
curtime()
```

The JDBC escape syntax `{fn curtime()}` is equivalent to the built-in syntax `CURRENT_TIME`. For more information, see CURRENT_TIME function.

hour

Returns the hour part of a time value.

```
hour(expression)
```

The JDBC escape syntax `{fn hour(expression)}` is equivalent to the built-in syntax `HOUR(expression)`. For more information, see HOUR function.

minute

Returns the minute part of a time value.

```
minute(expression)
```

The JDBC escape syntax `{fn minute(expression)}` is equivalent to the built-in syntax `MINUTE(expression)`. For more information, see MINUTE function.

month

Returns the month part of a date value.

```
month(expression)
```

The JDBC escape syntax `{fn month(expression)}` is equivalent to the built-in syntax `MONTH(expression)`. For more information, see MONTH function.

second

Returns the seconds part of a time value.

```
second(expression)
```

The JDBC escape syntax `{fn second(expression)}` is equivalent to the built-in syntax `SECOND(expression)`. For more information, see SECOND function.

TIMESTAMPADD

Use the `TIMESTAMPADD` function to add the value of an interval to a timestamp. The function applies the integer to the specified timestamp based on the interval type and returns the sum as a new timestamp. You can subtract from the timestamp by using negative integers.

TIMESTAMPADD is a JDBC escaped function and is accessible only by using the JDBC escape function syntax.

```
TIMESTAMPADD( interval, integerExpression, timestampExpression )
```

To perform TIMESTAMPADD on dates and times, it is necessary to convert the dates and times to timestamps. Dates are converted to timestamps by putting 00:00:00.0 in the time-of-day fields. Times are converted to timestamps by putting the current date in the date fields.

Do not put a datetime column inside a timestamp arithmetic function in WHERE clauses, because the optimizer will not use any index on the column.

TIMESTAMPDIFF

Use the TIMESTAMPDIFF function to find the difference between two timestamp values at a specified interval. For example, the function can return the number of minutes between two specified timestamps.

The TIMESTAMPDIFF is a JDBC escaped function and is accessible only by using the JDBC escape function syntax.

```
TIMESTAMPDIFF( interval, timestampExpression1, timestampExpression2 )
```

To perform TIMESTAMPDIFF on dates and times, it is necessary to convert the dates and times to timestamps. Dates are converted to timestamps by putting 00:00:00.0 in the time-of-day fields. Times are converted to timestamps by putting the current date in the date fields.

Do not put a datetime column inside a timestamp arithmetic function in WHERE clauses, because the optimizer will not use any index on the column.

year

Returns the year part of a date value.

```
year(expression)
```

The JDBC escape syntax {fn year(*expression*)} is equivalent to the built-in syntax YEAR(*expression*). For more information, see YEAR function.

Valid intervals for TIMESTAMPADD and TIMESTAMPDIFF

The TIMESTAMPADD and TIMESTAMPDIFF functions are used to perform arithmetic with timestamps. These two functions use the following valid intervals for arithmetic operations:

- SQL_TSI_DAY
- SQL_TSI_FRAC_SECOND
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR

Examples for the TIMESTAMPADD and TIMESTAMPDIFF escape functions

To return a timestamp value one month later than the current timestamp, use the following syntax:

```
{fn TIMESTAMPADD( SQL_TSI_MONTH, 1, CURRENT_TIMESTAMP)}
```

To return the number of weeks between now and the specified time on January 1, 2008, use the following syntax:

```
{fn TIMESTAMPDIFF(SQL_TSI_WEEK, CURRENT_TIMESTAMP,
  timestamp('2008-01-01-12.00.00.000000'))}
```

System function

user

Returns the authorization identifier or name of the current user. If there is no current user, it returns APP.

```
user()
```

The JDBC escape syntax `{fn user()}` is equivalent to the built-in syntax `USER`. For more information, see USER function.

JDBC escape syntax for outer joins

Derby interprets the JDBC escape syntax for outer joins (and all join operations) as equivalent to the correct SQL syntax for outer joins or the appropriate join operation.

For information about join operations, see JOIN operations.

Syntax

```
{oj JOIN operations [JOIN operations ]* }
```

Equivalent to

```
JOIN operations [JOIN operations ]*
```

```
-- outer join
SELECT *
FROM
{oj Countries LEFT OUTER JOIN Cities ON
   (Countries.country_ISO_code=Cities.country_ISO_code)}
-- another join operation
SELECT *
FROM
{oj Countries JOIN Cities ON
 (Countries.country_ISO_code=Cities.country_ISO_code)}
-- a TableExpression can be a joinOperation. Therefore
-- you can have multiple join operations in a FROM clause
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
FROM {oj EMPLOYEE E INNER JOIN DEPARTMENT
INNER JOIN EMPLOYEE M ON MGRNO = M.EMPNO ON E.WORKDEPT = DEPTNO};
```

JDBC escape syntax for time formats

Derby interprets the JDBC escape syntax for time as equivalent to the correct SQL syntax for times. Derby also supports the ISO format of 8 characters (6 digits, and 2 decimal points).

Syntax

```
{t 'hh:mm:ss'}
```

Equivalent to

```
TIME 'hh:mm:ss'
```

Example

```
VALUES {t '20:00:03'}
```

JDBC escape syntax for date formats

Derby interprets the JDBC escape syntax for dates as equivalent to the correct SQL syntax for dates.

Syntax

```
{d 'yyyy-mm-dd'}
```

Equivalent to

```
DATE 'yyyy-mm-dd'
```

Example

```
VALUES {d '1995-12-19'}
```

JDBC escape syntax for timestamp formats

Derby interprets the JDBC escape syntax for timestamp as equivalent to the correct SQL syntax for timestamps. Derby also supports the ISO format of 23 characters (17 digits, 3 dashes, and 3 decimal points).

Syntax

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

Equivalent to

```
TIMESTAMP 'yyyy-mm-dd hh:mm:ss.f...'
```

The fractional portion of timestamp constants (.f...) can be omitted.

```
VALUES {ts '1999-01-09 20:11:11.123455'}
```

Setting attributes for the database connection URL

Derby allows you to supply a list of attributes to its database connection URL, which is a JDBC feature.

The attributes are specific to Derby.

You typically set attributes in a semicolon-separated list following the protocol and subprotocol. For information on how you set attributes, see Attributes of the Derby database connection URL. This chapter provides reference information only.

Note: Attributes are not parsed for correctness. If you pass in an incorrect attribute or corresponding value, it is simply ignored.

bootPassword=key attribute

Function
Specifies the key to use to :
- Encrypt a new database
- Configure an existing unencrypted database for encryption
- Boot an existing encrypted database

Specify an alphanumeric string that is at least eight characters long.

Combining with other attributes

When you create a new database, the *bootPassword=key* attribute must be combined with the *create=true* and *dataEncryption=true* attributes.

When you configure an existing unencrypted database for encryption, the *bootPassword=key* attribute must be combined with the dataEncryption=true attribute. For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

When you boot an existing encrypted database, no other attributes are necessary.

Examples

```
-- create a new, encrypted database
jdbc:derby:newDB;create=true;dataEncryption=true;
    bootPassword=cseveryPlace
-- configure an existing unencrypted database for encryption
jdbc:derby:salesdb;dataEncryption=true;bootPassword=cseveryPlace
-- boot an existing encrypted database
jdbc:derby:encryptedDB;bootPassword=cseveryPlace
```

collation=collation attribute

Function

The `collation` attribute is an optional attribute that specifies whether collation is based on the territory specified for the database or Unicode codepoint collation. The valid values for the `collation` attribute are: TERRITORY_BASED and UCS_BASIC. The default collation is Unicode codepoint collation (UCS_BASIC).

Restriction: The `collation` attribute can be specified only when you create a database. You cannot specify this attribute on an existing database or when you upgrade a database.

If you include the `collation` attribute with the value TERRITORY_BASED when you create the Derby database, the collation is based on the language and country codes that you specify with the `territory` attribute.

If you do not specify the `territory` attribute when you create the database, Derby uses the `java.util.Locale.getDefault` method to determine the current value of the default locale for this instance of the Java Virtual Machine (JVM).

Note: The `collation` attribute applies only to user-defined tables. The system tables use the Unicode codepoint collation.

Example

The following example shows the URL to create the `MexicanDB` database. The territory attribute specifies Spanish for the language code and Mexico for the country code. The `collation` attribute specifies that the collation for the database is territory based.

```
jdbc:derby:MexicanDB;create=true;territory=es_MX;collation=TERRITORY_BASED
```

For information on how Derby handles collation, see "Character-based collation in Derby" in the *Java DB Developer's Guide*

create=true attribute

Function

Creates the standard database specified within the database connection URL Derby system and then connects to it. If the database cannot be created, the error appears in the error log and the connection attempt fails with an *SQLException* indicating that the database cannot be found.

If the database already exists, creates a connection to the existing database and an *SQLWarning* is issued.

JDBC does not remove the database on failure to connect at create time if failure occurs after the database call occurs. If a database connection URL used *create=true* and the connection fails to be created, check for the database directory. If it exists, remove it and its contents before the next attempt to create the database.

Database owner

When the database is created, the current authorization identifier becomes the database owner (see the *user=userName attribute*). If authentication and SQL authorization are both enabled (see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide*), only the database owner can shut down the database, encrypt it, reencrypt it with a new boot password or new encryption key, or perform a full upgrade. If authentication is not enabled, and no user is supplied, the database owner defaults to "APP", which is also the name of the default schema (see SET SCHEMA statement).

Combining with other attributes

You must specify a *databaseName* (after the subprotocol in the database connection URL) or a *databaseName=nameofDatabase* attribute with this attribute.

You can combine this attribute with other attributes. To specify a territory when creating a database, use the *territory=ll_CC* attribute.
Note: If you specify *create=true* and the database already exists, an *SQLWarning* is raised.

```
jdbc:derby:sampleDB;create=true
```

```
jdbc:derby:;databaseName=newDB;create=true;
```

createFrom=path attribute

Function

You can specify the *createFrom=path* attribute in the boot time connection URL to create a database using a full backup at a specified location. If there is a database with the same name in derby.system.home, an error will occur and the existing database will be left intact. If there is not an existing database with the same name in the current derby.system.home location, the whole database is copied from the backup location to the derby.system.home location and started.

The Log files are copied to the default location. The logDevice attribute can be used in conjunction with *createFrom=path* to store logs in a different location. With *createFrom=path* you do not need to copy the individual log files to the log directory.

For more information about using this attribute, see "Creating a database from a backup copy" in the *Java DB Server and Administration Guide*.

Combining with other attributes

Do not combine this attribute with *rollForwardRecoveryFrom*, *restoreFrom*, or *create*.

```
URL: jdbc:derby:wombat;createFrom=d:/backup/wombat
```

databaseName=nameofDatabase attribute

Function

Specifies a database name for a connection; it can be used instead of specifying the database name in after the subprotocol.

For example, these URL (and Properties object) combinations are equivalent:
- *jdbc:derby:toursDB*
- *jdbc:derby:;databaseName=toursDB*
- *jdbc:derby:*(with a property *databaseName* and its value set to *toursDB* in the *Properties* object passed into a connection request)

If the database name is specified both in the URL (as a subname) and as an attribute, the database name set as the subname has priority. For example, the following database connection URL connects to *toursDB*:

```
jdbc:derby:toursDB;databaseName=flightsDB
```

Allowing the database name to be set as an attribute allows the *getPropertyInfo* method to return a list of choices for the database name based on the set of databases known to Derby. For more information, see *java.sql.Driver.getPropertyInfo method.*

Combining with other attributes

You can combine this attribute with all other attributes.

```
jdbc:derby:;databaseName=newDB;create=true
```

dataEncryption=true attribute

Function

Specifies data encryption on disk for a new database or to configure an existing unencrypted database for encryption. For information about data encryption, see "Encrypting databases on disk" in the *Java DB Developer's Guide*.

Combining with other attributes

The *dataEncryption* attribute must be combined with the *bootPassword=key* attribute or the newEncryptionKey=key attribute. You have the option of also specifying the *encryptionProvider=providerName* and *encryptionAlgorithm=algorithm* attributes.

For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption. See also "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Examples

```
 -- encrypt a new database
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    bootPassword=cLo4u922sc23aPe
 -- configure an existing unencrypted database for encryption
   jdbc:derby:salesdb;dataEncryption=true;bootPassword=cLo4u922sc23aPe
```

encryptionKey=key attribute

Function

Specifies the external key to use to:
- Encrypt a new database
- Configure an existing unencrypted database for encryption
- Boot an existing encrypted database

Your application must provide the encryption key.

Combining with other attributes

When creating a new database, you must combine the *encryptionKey* attribute with the *create=true* and *dataEncryption=true* attributes.

When you configure an existing unencrypted database for encryption, the *encryptionKey* attribute must be combined with the *dataEncryption=true* attribute. For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

When booting an existing encrypted database, you must also specify the *encryptionAlgorithm* attribute if the algorithm that was used when the database was created is not the default algorithm.

The default encryption algorithm used by Derby is DES/CBC/NoPadding.

Examples

Example of a JDBC URL that creates a new encrypted database:

```
jdbc:derby:newDB;create=true;dataEncryption=true;
   encryptionAlgorithm=DES/CBC/NoPadding;encryptionKey=6162636465666768
```

Example of a JDBC URL that configures an existing unencrypted database for encryption:

```
jdbc:derby:salesdb;dataEncryption=true;encryptionKey=6162636465666768
```

Example of a JDBC URL that boots an encrypted database:

```
jdbc:derby:encryptedDB;encryptionKey=6162636465666768
```

encryptionProvider=providerName attribute

Function

Specifies the provider for data encryption. For information about data encryption, see "Encrypting databases on disk" in the *Java DB Developer's Guide*.

If this attribute is not specified, the default encryption provider is the one included in the JVM that you are using.

Combining with other attributes

The *encryptionProvider* attribute must be combined with the *bootPassword=key and dataEncryption=true* attributes. You can also specify the *encryptionAlgorithm=algorithm* attribute.

For an existing, unencrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption or reencryption. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Examples

```
-- create a new, encrypted database
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
-- configure an existing database for encryption
    jdbc:derby:salesdb;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
```

encryptionAlgorithm=algorithm attribute

Function

Specifies the algorithm for data encryption.

Use the Java conventions when you specify the algorithm, for example:

```
algorithmName/feedbackMode/padding
```

The only padding type that is allowed with Derby is *NoPadding*.

If no encryption algorithm is specified, the default value is *DES/CBC/NoPadding*.

For information about data encryption, see "Encrypting databases on disk" in the *Java DB Developer's Guide*.

Combining with other attributes

The *encryptionAlgorithm* attribute must be combined with the *bootPassword=key* attribute and the dataEncryption=true attribute. You have the option of also specifying the *encryptionProvider=providerName* attribute to specify the encryption provider of the algorithm.

For an existing database for which authentication and SQL authorization are both enabled, only the database owner can perform encryption or reencryption. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Examples

```
 -- encrypt a new database
    jdbc:derby:encryptedDB;create=true;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
-- configure an existing database for encryption
    jdbc:derby:salesdb;dataEncryption=true;
    encryptionProvider=com.sun.crypto.provider.SunJCE;
    encryptionAlgorithm=DESede/CBC/NoPadding;
    bootPassword=cLo4u922sc23aPe
```

Note: If the specified provider does not support the specified algorithm, Derby returns an exception.

failover=true attribute

Function

Stops database replication on the slave system and converts the slave database into a normal database.

If you specify the *failover=true* attribute on the master, the attribute sends the remaining log records to the slave instance and then sends a failover message to the slave. The replication functionality and the database are then shut down on the master system. If failover is successful, an exception with the error code XRE20 is thrown. Hence, when issued on the master, the *failover=true* attribute does not return a valid connection.

You may specify this attribute on the slave only if the network connection between the master and the slave is down.

When you specify this attribute on the slave, or when a failover message is sent as part of the execution of the *failover=true* attribute on the master, all transaction log chunks that have been received from the master are written to disk. The slave replication functionality is shut down, and the boot process of the database is allowed to complete. The database is now in a transaction consistent state, reflecting all transactions whose commit log records were received from the master. When issued on the slave, the *failover=true* command returns a valid connection.

The Derby instance where this command is issued must be serving the named database in replication mode.

For more information, see the topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. Authorization for the master database cannot be checked when the network connection is down, so the requirement that the user must be the database owner is not enforced.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- start failover from master using database name in subprotocol,
--   authorization
jdbc:derby:myDB;failover=true;user=mary;password=little88lamb
```

```
-- start failover using databaseName attribute, no security
jdbc:derby:;databaseName=myDB;failover=true;
```

logDevice=logDirectoryPath attribute

Function

The *logDirectoryPath* specifies the path to the directory on which to store the database log during database creation or restore. Even if specified as a relative path, the *logDirectoryPath* is stored internally as an absolute path.

For more information about using this attribute, see "Using the logDevice=logDirectoryPath attribute" in the *Java DB Server and Administration Guide*.

Combining with other attributes

Use in conjunction with *create*, *createFrom*, *restoreFrom*, *or rollForwardRecoveryFrom*.

```
jdbc:derby:newDB;create=true;logDevice=d:/newDBlog
```

newEncryptionKey=key attribute

Function

Specifies a new external encryption key for an encrypted database. All of the existing data in the database is encrypted using the new encryption key and any new data written to the database will use this key for encryption. For more information about this attribute, see "Encrypting databases with a new external encryption key" in the *Java DB Developer's Guide*.

Combining with other attributes

The *newEncryptionKey* attribute must be combined with the *encryptionKey=key* attribute.

You cannot change the encryption provider or the encryption algorithm when you use the *newEncryptionKey* attribute.

For an existing encrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform reencryption. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Example

```
-- specify a new encryption key for a database
jdbc:derby:salesdb;encryptionKey=6162636465666768;newEncryptionKey=6862636465666768
```

newBootPassword=newPassword attribute

Function

Specifies a new boot password for an encrypted database. A new encryption key is generated internally by the engine and the key is protected using the new boot password. The newly generated encryption key encrypts the database, including the existing data. For more information about this attribute, see "Encrypting databases with a new boot password" in the *Java DB Developer's Guide*.

Combining with other attributes

The *newBootPassword* attribute must be combined with the *bootPassword=key* attribute.

You cannot change the encryption provider or the encryption algorithm when you use the *newBootPassword* attribute.

For an existing encrypted database for which authentication and SQL authorization are both enabled, only the database owner can perform reencryption. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Example

```
-- specify a new boot password for a database
jdbc:derby:salesdb;bootPassword=abc1234xyz;newBootPassword=new1234xyz
```

password=userPassword attribute

Function

A valid password for the given user name.

Combining with other attributes

Use in conjunction with the *user=userName* attribute.

```
jdbc:derby:toursDB;user=jack;password=upTheHill
```

restoreFrom=path attribute

Function

You can specify the *restoreFrom=path* attribute in the boot time connection URL to restore a database using a full backup from the specified location. If a database with the same name exists in the derby.system.home location, the whole database is deleted, copied from the backup location, and then restarted.

The log files are copied to the same location they were in when the backup was taken. The logDevice attribute can be used in conjunction with *restoreFrom=path* to store logs in a different location.

For more information about using this attribute, see "Restoring a database from a backup copy" in the *Java DB Server and Administration Guide*.

Combining with other attributes

Do not combine this attribute with *createFrom*, *rollForwardRecoveryFrom*, or *create*.

```
URL: jdbc:derby:wombat;restoreFrom=d:/backup/wombat
```

rollForwardRecoveryFrom=path attribute

Function

You can specify the *rollForwardRecoveryFrom=path* in the boot time URL to restore the database using a backup copy and perform rollforward recovery using archived and active logs.

To restore a database using rollforward recovery, you must already have a backup copy of the database, all the archived logs since then, and the active log files. All the log files should be in the database log directory.

After a database is restored from full backup, transactions from the online archived logs and the active logs are replayed.

For more information about using this attribute, see "Roll-forward recovery" in the *Java DB Server and Administration Guide*.

Combining with other attributes

Do not combine this attribute with *createFrom*, *restoreFrom*, or *create*.

```
URL: jdbc:derby:wombat;rollForwardRecoveryFrom=d:/backup/wombat
```

securityMechanism=value attribute

Function

Specifies a security mechanism for client access to the Network Server. The *value* is numeric.

Valid numeric values are:

- **8**, which specifies Strong Password Substitute security. If you specify this mechanism, a strong password substitute is generated and used to authenticate the user with the network server. The original password is never sent in any form across the network.
- **9**, which specifies Encrypted UserID and Encrypted Password security. If you specify this mechanism, both the user ID and the password are encrypted. See "Enabling the encrypted user ID and password security mechanism" in the *Java DB Server and Administration Guide* for additional requirements for the use of this security mechanism.
- **3**, which specifies Clear Text Password security. Clear Text Password security is the default if you do not specify the *securityMechanism* attribute and you specify both the *user=userName* and *password=userPassword* attributes.
- **4**, which specifies User Only security. User Only security is the default if you do not specify the *securityMechanism* attribute and you specify the *user=userName* attribute but not the *password=userPassword* attribute.

Combining with other attributes

The *securityMechanism* attribute must be combined with the *user=userName* attribute.

Example

```
-- specify Strong Password Substitute security
jdbc:derby://localhost/
mydb;user=myuser;password=mypassword;securityMechanism=8
```

shutdown=true attribute

Function

Shuts down the specified database if you specify a *databaseName*. (Reconnecting to the database reboots the database.) For a database for which authentication and SQL authorization are both enabled, only the database owner can perform shutdown of that database. Please see "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Shuts down the entire Derby system if and only if you do not specify a *databaseName*.

When you are shutting down a single database, it lets Derby perform a final checkpoint on the database.

When you are shutting down a system, it lets Derby perform a final checkpoint on all system databases, deregister the JDBC driver, and shut down within the JVM before the JVM exits. A successful shutdown always results in an *SQLException* indicating that

Derby has shut down and that there is no connection. Once Derby is shut down, you can restart it by reloading the driver. For details on restarting Derby, see "Shutting down the system" in the *Java DB Developer's Guide*.

Checkpointing means writing all data and transaction information to disk so that no recovery needs to be performed at the next connection.

Used to shut down the entire system only when it is embedded in an application.
Note: Any request to the *DriverManager* with a *shutdown=true* attribute raises an exception.

```
-- shuts down entire system
jdbc:derby:;shutdown=true
-- shuts down salesDB (authentication not enabled)
jdbc:derby:salesDB;shutdown=true
```

slaveHost=hostname attribute

Function

Specifies the system that will serve as the slave for database replication.

For more information, see the topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

This attribute must be specified in conjunction with the *startMaster=true* attribute. It may be specified in conjunction with the *startSlave=true* attribute; if it is not, the default value is `localhost`.

This attribute may be specified only in conjunction with the other attributes permitted with the *startMaster=true* and *startSlave=true* attributes.

Examples

For examples, see *startMaster=true* and *startSlave=true*.

slavePort=portValue attribute

Function

Specifies the port that the slave system will use in database replication.

For more information, see the topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

This attribute may be specified in conjunction with the *startMaster=true* attribute and the *startSlave=true* attribute. If it is not specified, the default port value is `4851`.

This attribute may be specified only in conjunction with the other attributes permitted with the *startMaster=true* and *startSlave=true* attributes.

Examples

For examples, see *startMaster=true* and *startSlave=true*.

startMaster=true attribute

Function

Starts replication of a database in master mode. Before you specify this attribute, you must cleanly shut down the database on the master system, perform a file system copy of the database to the slave system, and specify the *startSlave=true* attribute. For details, see the topic "Starting and running replication" under "Replicating databases" in the *Java DB Server and Administration Guide*.

If the master database is already booted and any unlogged operations are running when the user specifies *startMaster=true*, the attempt to start the master fails and an error message appears.

For more information on replication, see the other topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

You must specify this attribute in conjunction with the *slaveHost=hostname* attribute. You may also specify this attribute in conjunction with the *slavePort=portValue* attribute. If you do not specify the *slavePort=portValue* attribute, the default port value is 4851.

If authentication or authorization is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If authorization is turned on, the user must be the database owner.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- start master using database name in subprotocol, default slave
--  port, authorization
jdbc:derby:myDB;startMaster=true;slaveHost=elsewhere;user=mary;
   password=little88lamb
```

```
-- start master using databaseName attribute, non-default slave
--  port, no security
jdbc:derby:;databaseName=myDB;startMaster=true;slaveHost=elsewhere;
   slavePort=4852
```

startSlave=true attribute

Function

Starts replication of a database in slave mode. Before you specify this attribute, you must cleanly shut down the database on the master system and then perform a file system copy of the database to the slave system.

The *startSlave=true* attribute does the following:

1. Partially boots the specified database
2. Starts to listen on the specified port and accepts a connection from the master
3. Hangs until the master has connected to it
4. Reports the startup status to the caller (whether it has started, and if not, why not)
5. Continually receives chunks of the transaction log from the master and applies the operations in the transaction log to the slave database

If replication is started successfully, an exception with the error code XRE08 is thrown. Hence, the *startSlave=true* attribute does not return a valid connection.

For more information, see the topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

You may specify this attribute in conjunction with the *slaveHost=hostname* attribute. If you do not specify the *slaveHost=hostname* attribute, the default value is `localhost`.

You may also specify this attribute in conjunction with the *slavePort=portValue* attribute. If you do not specify the *slavePort=portValue* attribute, the default port value is 4851.

If authentication or authorization is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If authorization is turned on, the user must be the database owner.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- start slave using database name in subprotocol, default slave host
--    and port, authorization
jdbc:derby:myDB;startslave=true;user=mary;password=little88lamb
```

```
-- start slave using databaseName attribute, non-default slave host
--   and port, no security
jdbc:derby:;databaseName=myDB;startSlave=true;slaveHost=localhost;
   slavePort=4852
```

stopMaster=true attribute

Function

Stops database replication on the master system. This attribute sends a stop-slave message to the slave system if the network connection is working. Then it shuts down all replication-related functionality, without shutting down the specified database.

The Derby instance where this attribute is specified must be the replication master for the specified database.

For more information, see the topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

If authentication or authorization is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. If authorization is turned on, the user must be the database owner.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- stop master using database name in subprotocol, authorization
jdbc:derby:myDB;stopMaster=true;user=mary;password=little88lamb
```

```
-- stop master using databaseName attribute, no security
jdbc:derby:;databaseName=myDB;stopMaster=true;
```

stopSlave=true attribute

Function

Stops database replication on the slave system.

You can specify this connection URL attribute only if the network connection between the master and slave systems is down. If the network connection is working, the slave system accepts commands only from the master, so you must specify the *stopMaster=true* attribute on the master system to stop replication on both the master and slave systems.

When this attribute is specified, or when a stop-slave message is sent as part of the execution of the *stopMaster=true* attribute, all transaction log chunks that have been received from the master are written to disk. Both the slave replication functionality and the database are then shut down.

The Derby instance where this attribute is specified must be serving the specified database in replication slave mode.

For more information, see the topics under "Replicating databases" in the *Java DB Server and Administration Guide*.

Combining with other attributes

You must specify a database name in the connection URL, either in the subprotocol or by using the *databaseName=nameofDatabase* attribute.

If authentication is turned on, you must also specify this attribute in conjunction with the *user=userName* and *password=userPassword* attributes. Authorization for the master database cannot be checked when the network connection is down, so the requirement that the user must be the database owner is not enforced.

You may not specify this attribute in conjunction with any attributes not mentioned in this section.

Examples

```
-- stop slave from master using database name in subprotocol,
--   authorization
jdbc:derby:myDB;stopSlave=true;user=mary;password=little88lamb
```

```
-- stop slave using databaseName attribute, no security
jdbc:derby:;databaseName=myDB;stopSlave=true;
```

territory=ll_CC attribute

Function

When creating or upgrading a database, use this attribute to associate a non-default territory with the database. Setting the *territory* attribute overrides the default system territory for that database. The default system territory is found using *java.util.Locale.getDefault()*.

Specify a territory in the form *ll_CC*, where *ll* is the two-letter language code, and *CC* is the two-letter country code.

Language codes consist of a pair of lowercase letters that conform to ISO-639.

Table 100. Sample Language Codes

Language Code	Description
de	German

Language Code	Description
en	English
es	Spanish
ja	Japanese

To see a full list of ISO-639 codes, go to *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*.

Country codes consist of two uppercase letters that conform to ISO-3166.

Table 101. Sample Country Codes

Country Code	Description
DE	Germany
US	United States
ES	Spain
MX	Mexico
JP	Japan

A copy of ISO-3166 can be found at *http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html*.

Combining with other attributes

The *territory* attribute is used only when creating a database.

In the following example, the new database has a territory of Spanish language and Mexican nationality.

```
jdbc:derby:MexicanDB;create=true;territory=es_MX
```

You can use the `collation` attribute with the `territory` attribute to specify that collation is based on the territory instead of based on Unicode codepoint collation,

traceDirectory=path attribute

Function

Specifies a directory to which the Derby Network Client will send JDBC trace information. If the program or session has multiple connections, the Network Client creates a separate file for each connection. By default, the files are named `_driver_0`, `_driver_1`, and so on. Use the *traceFile=path* attribute to specify a file name for the trace file.

If the directory does not exist, Derby issues an error message. If you do not specify an absolute path name, the directory is assumed to be relative to the current directory.

For more information about tracing, see "Network client tracing" in the *Java DB Server and Administration Guide*. See *traceFile=path*, *traceFileAppend=true*, and *traceLevel=value* for other attributes related to tracing.

Combining with other attributes

You can combine this attribute with other attributes.

Examples

```
-- enable tracing on an existing database that will have multiple
 connections
jdbc:derby://localhost:1527/mydb;traceDirectory=/home/mydir/mydbtracedir
-- specify a trace file name within the directory
jdbc:derby://localhost:1527/mydb;traceDirectory=/home/mydir/
mydbtracedir;traceFile=trace.out
-- append to the default trace file
jdbc:derby://localhost:1527/mydb;traceDirectory=/home/mydir/
mydbtracedir;traceFileAppend=true
```

traceFile=path attribute

Function

Specifies a file to which the Derby Network Client will send JDBC trace information. If you do not specify an absolute path name, the file is placed in the *derby.system.home* directory (see "Defining the system directory" in the *Java DB Developer's Guide* for details).

If you specify both *traceFile=path* and *traceFileAppend=true*, the output is appended to the specified file, if it exists. If you specify *traceFile=path* but do not specify *traceFileAppend=true*, any previous version of the file of the file is overwritten.

For more information about tracing, see "Network client tracing" in the *Java DB Server and Administration Guide*. See *traceDirectory=path* and *traceLevel=value* for other attributes related to tracing.

Combining with other attributes

You can combine this attribute with other attributes.

Example

```
-- enable tracing on a new database
jdbc:derby://localhost:1527/mydb;create=true;traceFile=trace.out
```

traceFileAppend=true attribute

Function

Specifies that the Derby Network Client should append JDBC trace information to a trace file. The file can be specified by the *traceFile=path* attribute. If you do not specify a trace file but you specify the *traceDirectory=path* attribute, the trace information is appended to the default file. If you do not specify *traceFileAppend=true*, any previous version of the trace file is overwritten.

For more information about tracing, see "Network client tracing" in the *Java DB Server and Administration Guide*. See *traceDirectory=path* and *traceLevel=value* for other attributes related to tracing.

Combining with other attributes

This attribute must be specified in conjunction with either the *traceFile=path* attribute or the *traceDirectory=path* attribute. You can also combine this attribute with other attributes.

Example

```
-- enable tracing on an existing database, appending to the
--   specified file
jdbc:derby://localhost:1527/mydb;traceFile=trace.out;traceFileAppend=true
-- enable tracing on an existing database, appending to the default file
--   within the specified directory, relative to the Derby home directory
```

```
jdbc:derby://localhost:1527/
mydb;traceDirectory=mytracedir;traceFileAppend=true
```

traceLevel=value attribute

Function

If tracing is enabled, specifies the level of tracing to be used by the Derby Network Client. The *value* is numeric. If you do not specify a trace level, the default is TRACE_ALL.

For more information about tracing, see "Network client tracing" in the *Java DB Server and Administration Guide*. See *traceFile=path*, *traceFileAppend=true*, and *traceDirectory=path* for other attributes related to tracing.

Tracing levels

The following table shows the available tracing levels and their values.

Table 102. Available tracing levels and values

Trace level	Hex value	Decimal value
org.apache.derby.jdbc.ClientDataSource.TRACE_NONE	0x0	0
org.apache.derby.jdbc.ClientDataSource.TRACE_CONNECTION_CALL	0x1	1
org.apache.derby.jdbc.ClientDataSource.TRACE_STATEMENT_CALLS	0x2	2
org.apache.derby.jdbc.ClientDataSource.TRACE_RESULT_SET_CALL	0x4	4
org.apache.derby.jdbc.ClientDataSource.TRACE_DRIVER_CONFIGUR	0x10	16
org.apache.derby.jdbc.ClientDataSource.TRACE_CONNECTS	0x20	32
org.apache.derby.jdbc.ClientDataSource.TRACE_PROTOCOL_FLOWS	0x40	64
org.apache.derby.jdbc.ClientDataSource.TRACE_RESULT_SET_META	0x80	128
org.apache.derby.jdbc.ClientDataSource.TRACE_PARAMETER_META	0x100	256
org.apache.derby.jdbc.ClientDataSource.TRACE_DIAGNOSTICS	0x200	512
org.apache.derby.jdbc.ClientDataSource.TRACE_XA_CALLS	0x800	2048
org.apache.derby.jdbc.ClientDataSource.TRACE_ALL	0xFFF	-1

To specify more than one trace level, use one of the following techniques:

- If you are using the `ij` tool, add the decimal values together and specify the sum. For example, to trace both PROTOCOL flows and connection calls, add the values for TRACE_PROTOCOL_FLOWS (64) and TRACE_CONNECTION_CALLS (1). Specify the sum, the value 65.
- If you are running a JDBC program, do one of the following:
  - Use bitwise OR operators (|) with two or more trace values. For example, to trace protocol flows and connection calls, specify this value for traceLevel:

    ```
    TRACE_PROTOCOL_FLOWS | TRACE_CONNECTION_CALLS
    ```
  - Use a bitwise complement operator (~) with a trace value to specify all except a certain trace. For example, to trace everything except protocol flows, specify this value for traceLevel:

    ```
    ~TRACE_PROTOCOL_FLOWS
    ```

Combining with other attributes

If you specify this attribute, you must also specify either the *traceFile=path* attribute or the *traceDirectory=path* attribute.

Example

```
-- enable tracing on a new database
jdbc:derby://localhost:1527/
mydb;create=true;traceFile=trace.out;traceLevel=65
```

upgrade=true attribute

Function

Upgrades a database that was created using an earlier version of Derby to the current version of Derby, then connects to it. If the database does not exist, an error appears in the error log and the connection attempt fails with an *SQLException* indicating that the database cannot be found.

This operation performs a full upgrade, as defined in "Upgrading a database" in the *Java DB Developer's Guide*. For more information about upgrades, see the other topics under "Upgrades" in the *Java DB Developer's Guide*.

For a database for which authentication and SQL authorization are both enabled, only the database owner can perform a full upgrade. See also "Enabling user authentication" and "Setting the SQL standard authorization mode" in the *Java DB Developer's Guide* for more information.

Combining with other attributes

You must specify a *databaseName* (after the subprotocol in the database connection URL) or a *databaseName=nameofDatabase* attribute with this attribute.

You cannot combine this attribute with the *collation* or *territory=ll_CC* attributes.

```
jdbc:derby:sampleDB;upgrade=true

jdbc:derby:;databaseName=sampleDB;upgrade=true;
```

user=userName attribute

Specifies a valid user name for the system, specified with a password. A valid user name and password are required when user authentication is turned on.

Combining with other attributes

Use in conjunction with the *password=userPassword* attribute.

The following database connection URL connects the user jill to *toursDB*:

```
jdbc:derby:toursDB;user=jill;password=toFetchAPail
```

ssl=sslMode attribute

Function

Specifies the SSL mode of the client. The *sslMode* can be `basic`, `peerAuthentication`, or `off` (the default). See *"Network encryption and authentication with SSL/TLS"* in the *Java DB Server and Administration Guide* for details.

Combining with other attributes

May be combined with all other attributes.

Example

Connecting to `mydb` with basic SSL encryption:

```
jdbc:derby://localhost/mydb;ssl=basic
```

Creating a connection without specifying attributes

If no attributes are specified, you must specify a *databaseName.*

Derby opens a connection to an existing database with that name in the current system directory. If the database does not exist, the connection attempt returns an *SQLException* indicating that the database cannot be found.

```
jdbc:derby:mydb
```

Derby property reference

This section provides reference information on Derby properties. For information on using these properties, see "Working with Derby properties" in the *Java DB Developer's Guide*.

Scope of Derby properties

A property in Derby belongs to one or more of these scopes:

- *system-wide*

  System-wide properties apply to an entire system, including all its databases and tables if applicable.

  - Set programmatically

    System-wide properties set programmatically have precedence over database-wide properties and system-wide properties set in the *derby.properties* file.
  - Set in the *derby.properties* file

    The *derby.properties* file is an optional file that can be created to set properties at the system level when the Derby driver is loaded. Derby looks for this file in the directory defined by the *derby.system.home* property. Any property except *derby.system.home* can be set by including it in the *derby.properties* file.
- *database-wide*

  A database-wide property is stored in a database and is valid for that specific database only.

Note: Database-wide properties are stored in the database and are simpler and safer for deployment. System-wide properties can be more practical during the development process.

For more information about scopes, precedence, and persistence, see "Properties overview" in the *Java DB Developer's Guide*.

Dynamic and static properties

Most properties are dynamic; that means you can set them while Derby is running, and their values change without requiring a reboot of Derby. In some cases, this change takes place immediately; in some cases, it takes place at the next connection.

Some properties are static, which means changes to their values will not take effect while Derby is running. You must restart or set them before (or while) starting Derby.

Note: Properties set in the *derby.properties* file and on the command line of the application that boots Derby are *always* static, because Derby reads this file and those parameters only at startup.

Only properties set in the following ways have the potential to be dynamic:

- As database-wide properties
- As system-wide properties via a *Properties* object in the application in which the Derby engine is embedded

Derby properties

The Derby properties are used for configuring the system and database, as well as for diagnostics such as logging statements, and monitoring and tracing locks.

The table Derby properties lists all the core Derby properties.

For information on how to set Derby properties, see "Setting Derby properties" in the *Java DB Developer's Guide*.

Note: When setting properties that have boolean values, be sure to trim extra spaces around the word *true*. Extra spaces around the word *true* cause the property to be set to false, regardless of its previous setting.

To disable or turn off a database-wide property setting, set its value to null. To determine the result of this action, recall that the search order for properties is as follows (as stated in "Precedence of properties" in the *Java DB Developer's Guide*).

1. [*] System-wide properties set programmatically (as a command-line option to the JVM when starting the application or within application code)
2. Database-wide properties
3. [*] System-wide properties set in the *derby.properties* file

[*] Not consulted if *derby.database.propertiesOnly* is set to true.

Setting the database-wide property to null has the effect of removing the property from the list of database properties and restoring the system property setting from *derby.properties* if there is one. As always, if no value can be determined from the search, the built-in default applies.

For example, the following code fragment turns off a previous database-wide setting of the *derby.database.fullAccessUsers* property:

```
Statement s = conn.createStatement();
s.executeUpdate("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
    "'derby.database.fullAccessUsers', null)");
```

If the property is a static one, the null setting does not take effect until you reboot the database. Moreover, the static property *derby.database.sqlAuthorization* cannot be disabled after it has been enabled, even with a reboot.

Derby properties summarizes the general Derby properties. In this table, S stands for system-wide, D stands for database-wide, and C indicates the value persists with newly created conglomerates. X means yes.

Table 103. Derby properties

Property	Scope	Dynamic
derby.authentication.ldap.searchAuthDN	S, D	
derby.authentication.ldap.searchAuthPW	S, D	
derby.authentication.ldap.searchBase	S, D	
derby.authentication.ldap.searchFilter	S, D	
derby.authentication.provider	S, D	
derby.authentication.server	S, D	
derby.connection.requireAuthentication	S, D	
derby.database.defaultConnectionMode	S, D	X*
derby.database.forceDatabaseLock	S	
derby.database.fullAccessUsers	S, D	X*

Property	Scope	Dynamic
derby.database.noAutoBoot	D	
derby.database.propertiesOnly	D	X
derby.database.readOnlyAccessUsers	S, D	X*
derby.database.sqlAuthorization	S, D	
derby.infolog.append	S	
derby.jdbc.xaTransactionTimeout	S, D	X
derby.language.logQueryPlan	S	
derby.language.logStatementText	S, D	
derby.locks.deadlockTimeout	S, D	X
derby.locks.deadlockTrace	S, D	X
derby.locks.escalationThreshold	S, D	X
derby.locks.monitor	S, D	X
derby.locks.waitTimeout	S, D	X
derby.replication.logBufferSize	S	
derby.replication.maxLogShippingInterval	S	
derby.replication.minLogShippingInterval	S	
derby.replication.verbose	S	
derby.storage.initialPages	C	
derby.storage.minimumRecordSize	S, D, C	X
derby.storage.pageCacheSize	S	
derby.storage.pageReservedSpace	S, D, C	X
derby.storage.pageSize	S, D, C	X
derby.storage.rowLocking	S, D	
derby.storage.tempDirectory	S, D	X
derby.stream.error.field	S	
derby.stream.error.file	S	
derby.stream.error.method	S	
derby.stream.error.logSeverityLevel	S	
derby.system.bootAll	S	
derby.system.durability	S	
derby.system.home	S	
derby.user.UserName	S, D	X

* See the main page for this property for information about when changes to the property are dynamic.

There are additional properties associated with the Derby tools. For more information about tool-specific properties, see the *Java DB Tools and Utilities Guide*.

derby.authentication.ldap.searchAuthDN

Function

Along with *derby.authentication.ldap.searchAuthPW*, this property indicates how Derby should bind with the LDAP directory server to do searches for user DN (distinguished name). This property specifies the DN; *derby.authentication.ldap.searchAuthPW* specifies the password to use for the search.

If these two properties are not specified, an anonymous search is performed if it is supported.

For more information about LDAP user authentication, see "LDAP directory service" in the *Java DB Developer's Guide*.

Syntax

```
derby.authentication.ldap.searchAuthDn=DN
```

Default

If not specified, an anonymous search is performed if it is supported.

Example

```
-- system-wide property
derby.authentication.ldap.searchAuthDn=
    cn=guest,o=ExampleSite.com
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchAuthDn',
    'cn=guest,o=ExampleSite.com')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.authentication.ldap.searchAuthPW

Function

Along with *derby.authentication.ldap.searchAuthDN*, indicates how Derby should bind with the directory server to do searches in order to retrieve a fully qualified user DN (distinguished name). This property specifies the password; *derby.authentication.ldap.searchAuthDN* specifies the DN to use for the search.

For more information about LDAP user authentication, see "LDAP directory service" in the *Java DB Developer's Guide*.

Default

If not specified, an anonymous search is performed if it is supported.

Example

```
-- system-wide property
derby.authentication.ldap.searchAuthPW=guestPassword
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchAuthPW',
    'guestPassword')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.authentication.ldap.searchBase

Function

Specifies the root DN of the point in your hierarchy from which to begin a guest or anonymous search for the user's DN. For example:

```
ou=people,o=ExampleSite.com
```

When using Netscape Directory Server, set this property to the root DN, the special entry to which access control does not apply.

For more information about LDAP user authentication, see the *Java DB Developer's Guide*.

Example

```
-- system-wide property
derby.authentication.ldap.searchBase=
    ou=people,o=ExampleSite.com
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchBase',
    'ou=people,o=ExampleSite.com')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.authentication.ldap.searchFilter

Function

Specifies the search filter to use to determine what constitutes a user (and other search predicate) for Derby searches for a full DN during user authentication.

If you set this property to *derby.user*, Derby looks for cached full DNs for users that you have defined with the *derby.user.UserName* property. For other users, Derby performs a search using the *default* search filter.

For more information about LDAP user authentication, see "LDAP directory service" in the *Java DB Developer's Guide*.

Syntax

```
derby.authentication.ldap.searchFilter=
    { searchFilter | derby.user)
```

Default

```
(&(objectClass=inetOrgPerson)(uid=userName))
```

Note: Derby automatically uses the filter you specify with ((uid=*userName*)) unless you include %USERNAME% in the definition. You might want to use %USERNAME% if your user DNs map the user name to something other than *uid* (for example, *user*).

Example

```
-- system-wide properties
derby.authentication.ldap.searchFilter=objectClass=person
## people in the marketing department
## Derby automatically adds (uid=<userName>)
derby.authentication.ldap.searchFilter=(&(ou=Marketing)
    (objectClass=person))
## all people but those in marketing
## Derby automatically adds (uid=<userName>)
derby.authentication.ldap.searchFilter=(&(!(ou=Marketing)
    (objectClass=person))
## map %USERNAME% to user, not uid
derby.authentication.ldap.searchFilter=(&((ou=People)
(user=%USERNAME%))
## cache user DNs locally and use the default for others
derby.authentication.ldap.searchFilter=derby.user

-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.ldap.searchFilter',
    'objectClass=person')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.authentication.provider

Function

Specifies the authentication provider for Derby user authentication.

Legal values include:

- LDAP

  An external LDAP directory service.
- BUILTIN

  Derby's simple internal user authentication repository.
- A complete Java class name

  A user-defined class that provides user authentication.

When using an external authentication service provider (LDAP), you must also set:

- *derby.authentication.server*

When using LDAP, you can set other LDAP-specific properties. See also:

- *derby.authentication.ldap.searchAuthDN*
- *derby.authentication.ldap.searchAuthPW*
- *derby.authentication.ldap.searchFilter*
- *derby.authentication.ldap.searchBase*

Alternatively, you can write your own class to provide a different external authentication service. This class must implement the public interface *org.apache.derby.authentication.UserAuthenticator* and throw exceptions of the type *java.sql.SQLException* where appropriate. Using a user-defined class makes Derby adaptable to various naming and directory services. For example, the class could allow Derby to hook up to an existing user authentication service that uses any of the standard directory and naming service providers to JNDI.

To enable any Derby user authentication, you must set the *derby.connection.requireAuthentication* property to true.

For more information about user authentication, see the *Java DB Developer's Guide*.

Syntax

```
derby.authentication.provider={ LDAP | BUILTIN | classProviderName }
```

Default

BUILTIN

Example

```
-- system-wide property
derby.authentication.provider=LDAP
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.provider',
    'BUILTIN')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.authentication.server

Function

Specifies the location of the external directory service that provides user authentication for the Derby system as defined with derby.authentication.provider. For LDAP, specify the host name and port number.

The server must be known on the network.

For more information about external user authentication, see "External directory service" in the *Java DB Developer's Guide*.

Default

Not applicable. Note that if the protocol type is unspecified, it defaults to LDAP.

Syntax

```
derby.authentication.server=
[{ ldap: | ldaps: | nisplus: }]
[//]

{
    hostname [ :portnumber ]
 |
    nisServerName/nisDomain
}
```

Example

```
-- system-wide property
##LDAP example
derby.authentication.server=godfrey:9090
##LDAP example
derby.authentication.server=ldap://godfrey:9090
##LDAP example
derby.authentication.server=//godfrey:9090
##LDAP over SSL example
derby.authentication.server=ldaps://godfrey:636/
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.server',
```

```
    'godfrey:9090')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.connection.requireAuthentication

Function

Turns on user authentication for Derby.

When user authentication is turned on, a connection request must provide a valid user name and password.

Derby uses the type of user authentication specified with the *derby.authentication.provider* property.

For more information about user authentication, see "Working with user authentication" in the *Java DB Developer's Guide*.

Default

False.

By default, no user authentication is required.

Example

```
-- system-wide property
derby.connection.requireAuthentication=true
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.connection.requireAuthentication',
    'true')
```

Dynamic or static

Static. For system-wide properties, you must reboot Derby for the change to take effect. For database-wide properties, you must reboot the database for the change to take effect.

derby.database.defaultConnectionMode

Function

One of the user authorization properties.

Defines the default connection mode for users of the database or system for which this property is set. The possible values (which are case-insensitive) are:

- *noAccess*

  Disallows connections.
- *readOnlyAccess*

  Grants read-only connections.
- *fullAccess*

  Grants full access.

If the property is set to an invalid value, an exception is raised.

Note: It is possible to configure a database so that it cannot be changed (or even accessed) using this property. If you set this property to *noAccess* or *readOnlyAccess*, be sure to allow at least one user full access. See *derby.database.fullAccessUsers* and *derby.database.readOnlyAccessUsers.*

For more information about user authorization, see "User Authorization" in the *Java DB Developer's Guide*.

Syntax

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    '{ noAccess | readOnlyAccess | fullAccess}')
```

Example

```
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode', 'noAccess')
-- system-wide property
derby.database.defaultConnectionMode=noAccess
```

Default

fullAccess

Dynamic or static

Dynamic. Current connections are not affected, but all future connections are affected. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.forceDatabaseLock

Function

On some platforms, if set to true, prevents Derby from booting a database if a *db.lck* file is present in the database directory.

Derby attempts to prevent two JVMs from accessing a database at one time (and potentially corrupting it) with the use of a file called *db.lck* in the database directory. On some operating systems, the use of a lock file does not guarantee single access, and so Derby only issues a warning and might allow multiple JVM access even when the file is present. (For more information, see "Double-booting system behavior" in the *Java DB Developer's Guide*.)

Derby provides the property *derby.database.forceDatabaseLock* for use on platforms that do not provide the ability for Derby to guarantee single JVM access. By default, this property is set to false. When this property is set to true, if Derby finds the *db.lck* file when it attempts to boot the database, it throws an exception and does not boot the database.

Note: This situation can occur even when no other JVMs are accessing the database; in that case, remove the *db.lck* file by hand in order to boot the database. If the *db.lck* file is removed by hand while a JVM is still accessing a Derby database, there is no way for Derby to prevent a second VM from starting up and possibly corrupting the database. In this situation no warning message is logged to the error log.

Default

False.

Example

```
derby.database.forceDatabaseLock=true
```

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.database.fullAccessUsers

Function

One of the user authorization properties. Specifies a list of users to which full (read-write) access to a database is granted. The list consists of user names separated by commas. Do not put spaces after commas.

When set as a system property, specifies a list of users for which full access to all the databases in the system is granted.

See also *derby.database.readOnlyAccessUsers*.

A malformed list of user names raises an exception. Do not specify a user both with this property and in *derby.database.readOnlyAccessUsers*.

Note: User names, called authorization identifiers, follow the rules of *SQL92Identifiers* and can be delimited. Specifying a user name that does not follow these rules raises an exception.

For more information about user authorization, see "User Authorization" in the *Java DB Developer's Guide*.

Syntax

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers',
    'commaSeparatedlistOfUsers')
```

Example

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'dba,fred,peter')
--system-level property
derby.database.fullAccessUsers=dba,fred,peter
```

Dynamic or static

Dynamic. Current connections are not affected, but all future connections are affected. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.noAutoBoot

Function

Specifies that a database should not be automatically booted at startup time.

When this property is set to true, this database is booted only on the first connection. Otherwise, the database is booted at startup if the *derby.system.bootAll* property is set to true. See *derby.system.bootAll* for details.

Default

False.

Example

```
-- database-level property
```

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.noAutoBoot', 'true')
```

Scope

database-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.database.propertiesOnly

Function

When set to true, this property ensures that database-wide properties cannot be overridden by system-wide properties.

When this property is set to false, or not set, database-wide properties can be overridden by system-wide properties (see "Precedence of properties" in the *Java DB Developer's Guide*).

This property ensures that a database's environment cannot be modified by the environment in which it is booted.

This property can *never* be overridden by system properties.

Default

False.

Example

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.propertiesOnly','true')
```

Dynamic or static

This property is dynamic; if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.readOnlyAccessUsers

Function

One of the user authorization properties. Specifies a list of users to which read-only access to a database is granted. The list consists of user names separated by commas. Do not put spaces after commas.

When set as a system property, specifies a list of users for which read-only access to all the databases in the system is granted.

See also *derby.database.fullAccessUsers*.

A malformed list of user names raises an exception. Do not specify a user both in this property and in *derby.database.fullAccessUsers*.

Note: User names, called authorization identifiers, follow the rules of *SQL92Identifiers* and can be delimited. Specifying a user name that does not follow these rules raises an exception.

Syntax

```
-- database-level property
```

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers',
    'commaSeparatedListOfUsers')
```

Example

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers', 'ralph,guest')
-- system-level property
derby.database.readOnlyAccessUsers=ralph,guest
```

Dynamic or static

Dynamic. Current connection is not affected, but all future connections are affected. For information about dynamic changes to properties, see Dynamic and static properties.

derby.database.sqlAuthorization

Function

One of the user authorization properties.

Enables the SQL standard authorization mode for the database or system on which this property is set. The possible values are:

- *TRUE*

  SQL authorization for the database or system is enabled, which allows the use of GRANT and REVOKE statements.
- *FALSE*

  SQL authorization for the database or system is disabled. After this property is set to TRUE, the property cannot be set back to FALSE.

The values are not case-sensitive.

Note: If you set this property as a system property before you create the databases, all new databases will automatically have SQL authorization enabled. If the databases already exists, you can set this property only as a database property.

Derby uses the type of user authentication that is specified with the `derby.authentication.provider` property.

For more information about user authorization, see "User authorizations" in the *Java DB Developer's Guide*.

Example

```
-- system-wide property
derby.database.sqlAuthorization=true
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.sqlAuthorization', 'true');
```

Default

FALSE

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.infolog.append

Function

Specifies whether to append to or overwrite (delete and recreate) the *derby.log* file when the Derby engine is started. The *derby.log* file is used to record errors and other information. This information can help you debug problems within a system.

You can set this property even if the file does not yet exist; Derby creates the file.

Default

False.

By default, the file is deleted and then re-created.

Example

```
derby.infolog.append=true
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.jdbc.xaTransactionTimeout

Function

Specifies the default value of the XA transaction timeout that is used when a user either does not specify the XA transaction timeout or requests to use the default value. It is possible to use the `XAResource.setTransactionTimeout` method to specify the XA transaction timeout value for the global transaction.

A zero or negative value for this property means that the transaction timeout is not used.

Default

The transaction timeout is not used.

Example

```
-- system-wide property
derby.jdbc.xaTransactionTimeout=120
```

```
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.jdbc.xaTransactionTimeout', '120')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.language.logQueryPlan

Function

When this property is set to true, Derby writes the query plan information into the `derby.log` file for all executed queries. This information can help you debug problems within a system.

Example

```
derby.language.logQueryPlan=true
```

Default

False.

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.language.logStatementText

Function

When this property is set to true, Derby writes the text and parameter values of all executed statements to the information log at the beginning of execution. It also writes information about commits and rollbacks. Information includes the time and thread number.

This property is useful for debugging.

Example

```
derby.language.logStatementText=true
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.language.logStatementText', 'true')
```

Default

False.

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.locks.deadlockTimeout

Function

Determines the number of seconds after which Derby checks whether a transaction waiting to obtain a lock is involved in a deadlock. If a deadlock has occurred, and Derby chooses the transaction as a deadlock victim, Derby aborts the transaction. The transaction receives an *SQLException* of *SQLState* 40001. If the transaction is not chosen as the victim, it continues to wait for a lock if *derby.locks.waitTimeout* is set to a higher value than the value of *derby.locks.deadlockTimeout*.

If this property is set to a higher value than *derby.locks.waitTimeout*, no deadlock checking occurs. See *derby.locks.waitTimeout*.

For more information about deadlock checking, see "Deadlocks" in the *Java DB Developer's Guide*.

Default

20 seconds.

Example

```
derby.locks.deadlockTimeout=30
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.deadlockTimeout', '30')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.deadlockTrace

Function

Causes a detailed list of locks at the time of a deadlock or a timeout to be written to the error log (typically the file *derby.log*). For a deadlock, Derby describes the cycle of locks which caused the deadlock. For a timeout, Derby prints the entire lock list at the time of the timeout. This property is meaningful only if the *derby.locks.monitor* property is set to *true*.

Note: This level of debugging is intrusive: it can alter the timing of the application, reduce performance severely, and produce a large error log file. It should be used with care.

Default

False.

Example

```
-- system property
derby.locks.deadlockTrace=true

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.deadlockTrace', 'true')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.escalationThreshold

Function

Used by the Derby system at runtime in determining when to attempt to escalate locking for at least one of the tables involved in a transaction from row-level locking to table-level locking.

A large number of row locks use a lot of resources. If nearly all the rows are locked, it might be worth the slight decrease in concurrency to lock the entire table to avoid the large number of row locks.

For more information, see "Locking and performance" in *Tuning Java DB*.

It is useful to increase this value for large systems (such as enterprise-level servers, where there is more than 64 MB of memory), and to decrease it for very small systems (such as palmtops).

Syntax

```
derby.locks.escalationThreshold=numberOfLocks
```

Default

5000.

Minimum value

100.

Maximum value

2,147,483,647.

Example

```
-- system-wide property
derby.locks.escalationThreshold=1000
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.escalationThreshold',
    '1000')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.monitor

Function

Specifies that all deadlock errors are logged to the error log. If *derby.stream.error.logSeverityLevel* is set to ignore deadlock errors, *derby.locks.monitor* overrides it.

Default

False.

Example

```
-- system property
derby.locks.monitor=true

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.monitor', 'true')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.locks.waitTimeout

Function

Specifies the number of seconds after which Derby aborts a transaction when it is waiting for a lock. When Derby aborts (and rolls back) the transaction, the transaction receives an *SQLException* of *SQLState* 40XL1.

The time specified by this property is approximate.

A zero value for this property means that Derby aborts a transaction any time it cannot immediately obtain a lock that it requests.

A negative value for this property is equivalent to an infinite wait time; the transaction waits forever to obtain the lock.

If this property is set to a value greater than or equal to zero but less than the value of *derby.locks.deadlockTimeout*, Derby never performs any deadlock checking.

Default

60 seconds.

Example

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.locks.waitTimeout', '15')
derby.locks.waitTimeout=60
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.replication.logBufferSize

Function

Specifies the size of the replication log buffers in bytes. These buffers store the log on the master side before it is shipped to the slave. There is a total of 10 such buffers. Large buffers increase memory usage but reduce the chance that the buffers will fill up (in turn increasing response time for transactions on the master, as described in the failure situation "The master Derby instance is not able to send log data to the slave at the same pace as the log is generated" in the topic "Replication failure handling" in the *Java DB Server and Administration Guide*).

You can also use the properties *derby.replication.minLogShippingInterval* and *derby.replication.maxLogShippingInterval* to tune the rate at which the log is shipped from the master to the slave.

Minimum value

8192 (8 KB).

Maximum value

The maximum value is 1048576 (1 MB).

Default

32768 bytes (32KB).

Example

```
derby.replication.logBufferSize=65536
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.replication.maxLogShippingInterval

Function

Specifies, in milliseconds, the longest interval between two consecutive shipments of the transaction log from the master to the slave. This property provides a "soft" guarantee that the slave will not deviate more than this number of milliseconds from the master.

The value specified for the *derby.replication.maxLogShippingInterval* property must be at least ten times the value specified for the *derby.replication.minLogShippingInterval* property. If you set *derby.replication.maxLogShippingInterval* to a lower value, Derby

changes the *derby.replication.minLogShippingInterval* property value to the value of the *derby.replication.maxLogShippingInterval* property divided by 10.

Default

5000 milliseconds (5 seconds).

Example

```
derby.replication.maxLogShippingInterval=10000
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.replication.minLogShippingInterval

Function

Specifies, in milliseconds, the shortest interval between two consecutive shipments of the transaction log from the master to the slave.

The value specified for the *derby.replication.minLogShippingInterval* property must be no more than one-tenth the value specified for the *derby.replication.maxLogShippingInterval* property. If you set *derby.replication.minLogShippingInterval* to a higher value, Derby changes the *derby.replication.minLogShippingInterval* property value to the value of the *derby.replication.maxLogShippingInterval* property divided by 10.

Default

100 milliseconds.

Example

```
derby.replication.minLogShippingInterval=500
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.replication.verbose

Function

Specifies whether replication messages are written to the Derby log.

Default

True.

Example

```
derby.replication.verbose=false
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.storage.initialPages

Function

The on-disk size of a Derby table grows by one page at a time until eight pages of user data (or nine pages of total disk use; one is used for overhead) have been allocated. Then it will grow by eight pages at a time if possible.

A Derby table or index can be created with a number of pages already pre-allocated. To do so, specify the property prior to the CREATE TABLE or CREATE INDEX statement.

Define the number of user pages the table or index is to be created with. The purpose of this property is to preallocate a table or index of reasonable size if the user expects that a large amount of data will be inserted into the table or index. A table or index that has the pre-allocated pages will enjoy a small performance improvement over a table or index that has no pre-allocated pages when the data are loaded.

The total desired size of the table or index should be the following number of bytes:

```
(1 + derby.storage.initialPages) * derby.storage.pageSize
```

When you create a table or an index after setting this property, Derby attempts to preallocate the requested number of user pages. However, the operations do not fail even if they are unable to preallocate the requested number of pages, as long as they allocate at least one page.

Default

1 page.

Minimum value

The minimum number of *initialPages* is 1.

Maximum value

The maximum number of *initialPages* is 1000.

Example

```
-- system-wide property
derby.storage.initialPages=30

-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
'derby.storage.initialPages', '30')
```

derby.storage.minimumRecordSize

Function

Indicates the minimum user row size in bytes for on-disk database pages for tables when you are creating a table. This property ensures that there is enough room for a row to grow on a page when updated without having to overflow. This is generally most useful for VARCHAR and VARCHAR FOR BIT DATA data types and for tables that are updated a lot, in which the rows start small and grow due to updates. Reserving the space at the time of insertion minimizes row overflow due to updates, but it can result in wasted space. Set the property prior to issuing the CREATE TABLE statement.

See also *derby.storage.pageReservedSpace*.

Valid conglomerates

Tables only.

Default

12 bytes.

Minimum value

12 bytes.

Maximum value

derby.storage.pageSize * (1 - *derby.storage.pageReservedSpace*/100) " 100.

If you set this property to a value outside the legal range, Derby uses the default value.

Example

```
-- changing the default for the system
derby.storage.minimumRecordSize=128
-- changing the default for the database
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.minimumRecordSize',
    '128')
```

Dynamic or static

This property is dynamic; if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.storage.pageCacheSize

Function

Defines the size, in number of pages, of the data page cache in the database (data pages kept in memory).

The actual amount of memory the page cache will use depends on the following:

- The size of the cache, configured with the *derby.storage.pageCacheSize* property.
- The size of the pages, configured with the *derby.storage.pageSize* property. Derby automatically tunes for the database page size. If you have long columns, the default page size for the table is set to 32768 bytes. Otherwise, the default is 4096 bytes.
- Overhead, which varies with JVMs.

When increasing the size of the page cache, you typically have to allow more memory for the Java heap when starting the embedding application (taking into consideration, of course, the memory needs of the embedding application as well). For example, using the default page size of 4K, a page cache size of 2000 pages will require at least 8 MB of memory (and probably more, given the overhead).

For a simple application (no GUI), using the Sun 1.1.7 JVM on Windows NT and using the -mx96m option (which allows 96 MB for the Java heap), it is possible to have a page cache size of 10,000 pages (approximately 40 MB).

The minimum value is 40 pages. If you specify a lower value, Derby uses the default value.

Default

1000 pages.

Example

```
derby.storage.pageCacheSize=160
```

Dynamic or static

Static. You must reboot the system for the change to take effect.

derby.storage.pageReservedSpace

Function

Defines the percentage of space reserved for updates on an on-disk database page for tables only (not indexes); indicates the percentage of space to keep free on a page when inserting. Leaving reserved space on a page can minimize row overflow (and the associated performance hit) during updates. Once a page has been filled up to the reserved-space threshold, no new rows are allowed on the page. This reserved space is used only for rows that increase in size when updated, not for new inserts. Set this property prior to issuing the CREATE TABLE statement.

Regardless of the value of *derby.storage.pageReservedSpace*, an empty page always accepts at least one row.

Valid conglomerates

Tables only.

Default

20%.

Minimum value

The minimum value is 0% and the maximum is 100%. If you specify a value outside this range, Derby uses the default value of 20%.

Example

```
-- modifying the default for the system
derby.storage.pageReservedSpace=40
-- modifying the default for the database
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.pageReservedSpace',
    '40')
```

Dynamic or static

This property is dynamic: if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.storage.pageSize

Function

Defines the page size, in bytes, for on-disk database pages for tables or indexes used during table or index creation. Set this property prior to issuing the CREATE TABLE or CREATE INDEX statement. This value will be used for the lifetime of the newly created conglomerates.

Valid conglomerates

Tables and indexes, including the indexes created to enforce constraints.

Default

Derby automatically tunes for the database page size. If you have long columns, the default page size for the table is set to 32768 bytes. Otherwise, the default is 4096 bytes.

Valid values

Page size can only be one the following values: 4096, 8192, 16384, or 32768 bytes. If you specify an invalid value, Derby uses the default value.

Example

```
-- changing the default for the system
derby.storage.pageSize=8192
-- changing the default for the database
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.pageSize',
    '8192')
```

Dynamic or static

This property is dynamic; if you change it while Derby is running, the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

derby.storage.rowLocking

Function

If set to true, enables row-level locking. When you disable row-level locking, you use table-level locking.

Row-level locking uses more system resources but allows greater concurrency, which works better in multi-user systems. Table-level locking works best with single-user applications or read-only applications.

If you use row-level locking (the default), the system decides whether to use table-level locking or row-level locking for each table in each DML statement. In certain situations, the system might choose to escalate the locking scheme from row-level locking to table-level locking to improve performance.

For more information about locking, see "Locking and performance" in *Tuning Java DB*, and "Locking, concurrency, and isolation" in the *Java DB Developer's Guide*.

Default

True.

Example

```
-- system-wide property
derby.storage.rowLocking=false
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.rowLocking', 'false')
```

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.storage.tempDirectory

Function

Defines the location on disk for temporary file space needed by Derby for performing large sorts and deferred deletes and updates. (Temporary files are automatically deleted after use, and are removed when the database restarts after a crash.) The temporary directory named by this property will be created if it does not exist, but will not be deleted when the system shuts down. The path name specified by this property must have file separators that are appropriate to the current operating system.

This property allows databases located on read-only media to write temporary files to a writable location. If this property is not set, databases located on read-only media might get an error like the following:

```
ERROR XSDF1: Exception during creation
 of file  c:\databases\db\tmp\T887256591756.tmp
for container
ERROR XJ001: Java exception:
'a:\databases\db\tmp\T887256591756.tmp: java.io.IOException'.
```

This property moves the temporary directories for all databases being used by the Derby system. Derby makes temporary directories for each database under the directory referenced by this property. For example, if the property is set as follows:

```
derby.storage.tempDirectory=C:/Temp/dbtemp
```

the temporary directories for the databases in *C:\databases\db1* and *C:\databases\db2* will be in *C:\Temp\dbtemp\db1* and *C:\Temp\dbtemp\db2*, respectively.

The temporary files of two databases running concurrently with the same name (for example, *C:\databases\db1* and *E:\databases\db1*) will conflict with each other if the *derby.storage.tempDirectory* property is set. This will cause incorrect results, so users are advised to give databases unique names.

Default

A subdirectory named *tmp* under the database directory.

For example, if the database *db1* is stored in *C:\databases\db1*, the temporary files are created in *C:\databases\db1\tmp*.

Example

```
-- system-wide property
derby.storage.tempDirectory=c:/Temp/dbtemp
-- database-wide property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.storage.tempDirectory',
    'c:/Temp/dbtemp')
```

Dynamic or static

This property is static; you must restart Derby for a change to take effect.

derby.stream.error.field

Function

Specifies a static field that references a stream to which the error log is written. The field is specified using the fully qualified name of the class, then a dot (.) and then the field name. The field must be public and static. Its type can be either *java.io.OutputStream* or *java.io.Writer*.

The field is accessed once at Derby boot time, and the value is used until Derby is rebooted. If the field is null, the error stream defaults to the system error stream (*java.lang.System.err*).

If the field does not exist or is inaccessible, the error stream defaults to the system error stream. Derby will not call the *close()* method of the object obtained from the field.

Default

None.

Example

```
derby.stream.error.field=java.lang.System.err
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.stream.error.file

Function

Specifies name of the file to which the error log is written. If the file name is relative, it is taken as relative to the system directory.

If this property is set, the *derby.stream.error.method* and *derby.stream.error.field* properties are ignored.

Default

derby.log.

Example

```
derby.stream.error.file=error.txt
```

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.stream.error.method

Function

Specifies a static method that returns a stream to which the Derby error log is written.

Specify the method using the fully qualified name of the class, then a dot (.) and then the method name. The method must be public and static. Its return type can be either *java.io.OutputStream* or *java.io.Writer*. Derby will not call the *close()* method of the object returned by the method.

The method is called once at Derby boot time, and the return value is used for the lifetime of Derby. If the method returns null, the error stream defaults to the system error stream. If the method does not exist or is inaccessible, the error stream defaults to the system error stream (*java.lang.System.err*).

If the value of this property is set, the property *derby.stream.error.field* is ignored.

Default

Not set.

Example

```
derby.stream.error.method=java.sql.DriverManager.getLogStream
```

Scope

system-wide

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.stream.error.logSeverityLevel

Function

Specifies which errors are logged to the Derby error log (typically the *derby.log* file). In test environments, use the setting *derby.stream.error.logSeverityLevel=0* so that all problems are reported.

Any error raised in a Derby system is given a level of severity. This property indicates the minimum severity necessary for an error to appear in the error log. The severities are defined in the class *org.apache.derby.types.ExceptionSeverity*. The higher the number, the more severe the error.

- *20000*

  Errors that cause the statement to be rolled back, for example syntax errors and constraint violations.
- *30000*

  Errors that cause the transaction to be rolled back, for example deadlocks.
- *40000*

  Errors that cause the connection to be closed.
- *50000*

  Errors that shut down the Derby system.

Default

40000.

Example

```
// send errors of level 30000 and higher to the log
derby.stream.error.logSeverityLevel=30000
```

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.system.bootAll

Function

Specifies that all databases in the directory specified by the *derby.system.home* property should be automatically booted at startup time.

When this property is set to true, databases in the *derby.system.home* directory are booted at startup. Otherwise, databases are booted when you first connect to them.

You may want to use the *derby.system.bootAll* property to avoid a delay at first connection time. After a crash, a boot that requires recovery can take a long time, and you may want to perform this boot as soon as Derby is restarted.

You can set the *derby.database.noAutoBoot* property on a particular database if you want to prevent it from being automatically booted at startup. See *derby.database.noAutoBoot* for details.

Default

False.

Scope

system-wide

Example

```
derby.system.bootAll=true
```

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.system.durability

Function

This property changes the default durability of Derby to improve performance at the expense of consistency and durability of the database. The only valid supported case insensitive value is `test`. If this property is set to any value other than `test`, this property setting is ignored. When `derby.system.durability` is set to `test`, the store system will not force I/O synchronization calls for:

- The log file at each commit
- The log file before a data page is forced to disk
- Page allocation when a file is grown
- Data writes during checkpoints

While performance is improved, note that under these conditions, a commit no longer guarantees that the transaction's modification will survive a system crash or JVM termination, the database may not recover successfully upon restart, a near-full disk at runtime may cause unexpected errors, and the database may be in an inconsistent state.

If you boot the database with this property set to `test`, the following warning message is logged in the `derby.log` file:

```
WARNING: The database is booted with derby.system.durability=test.
In this mode, it is possible that database may not be able
to recover, committed transactions may be lost, and the database
may be in an inconsistent state. Please use this mode only when
these consequences are acceptable.
```

A similar message will appear in the `derby.log` file if the database was booted with `derby.system.durability=test` at any time previously.

Once the database is booted with `derby.system.durability=test`, there are no guarantees that the database is consistent.

Default

This property is ignored by default.

Supported values

The only supported value is `test`.

Example

```
derby.system.durability=test
```

Since this is a system property, you can set it in the `derby.properties` file or on the command line of the JVM when starting the application.

You might enable this property when using Derby as a test database where consistency or recoverability is not an issue.

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.system.home

Function

Specifies the Derby system directory, which is the directory that contains subdirectories holding databases that you create and the text file *derby.properties*.

If the system directory that you specify with *derby.system.home* does not exist at startup, Derby creates the directory automatically.

Default

Current directory (the value of the JVM system property *user.dir*).

If you do not explicitly set the *derby.system.home* property when starting Derby, the default is the directory in which Derby was started.

Note: You should always explicitly set the value of *derby.system.home*.

Example

```
-Dderby.system.home=C:\derby
```

Dynamic or static

This property is static; if you change it while Derby is running, the change does not take effect until you reboot.

derby.user.UserName

Function

Has two uses:

- Creates users and passwords when *derby.authentication.provider* is set to *BUILTIN*.
- Caches user DNs locally when *derby.authentication.provider* is set to *LDAP* and *derby.authentication.ldap.searchFilter* is set to *derby.user*.

Users and Passwords

This property creates valid clear-text users and passwords within Derby when the *derby.authentication.provider* property is set to *BUILTIN*. For information about users, see "Working with user authentication" in the *Java DB Developer's Guide*.

- *Database-Level Properties*

  When you create users with database-level properties, those users are available to the specified database only.

  You set the property once for each user. To delete a user, set that user's password to null.
- *System-Level Properties*

When you create users with system-level properties, those users are available to all databases in the system.

You set the value of this system-wide property once for each user, so you can set it several times. To delete a user, remove that user from the file.

You can define this property in the usual ways -- typically in the *derby.properties* file.

When a user name and its corresponding password are provided in the *DriverManager.getConnection* call, Derby validates them against the properties defined for the current system.

User names are *SQL92Identifiers* and can be delimited.

Syntax

```
derby.user.{UserName=Password} | UserName=userDN }
```

```
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.UserName',
    'Password | userDN')
```

Default

None.

Example

```
-- system-level property
derby.user.guest=java5w
```

```
derby.user.sa=Derby3x9

derby.user."!Amber"=java5w
-- database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.sa',

 'Derby3x9')
-- cache a userDN locally, database-level property
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.richard',
    'uid=richard, ou=People, o=ExampleSite.com')
```

Dynamic or static

Dynamic; the change takes effect immediately. For information about dynamic changes to properties, see Dynamic and static properties.

Caching user DNs

This property caches user DNs (distinguished names) locally when *derby.authentication.provider* is set to *LDAP* and *derby.authentication.ldap.searchFilter* is set to *derby.user*. When you provide a user DN with this property, Derby is able to avoid an LDAP search for that user's DN before authenticating. For those users without DNs defined with this property, Derby performs a search using the default value of *derby.authentication.ldap.searchFilter*.

J2EE Compliance: Java Transaction API and javax.sql Interfaces

J2EE, or the Java 2 Platform, Enterprise Edition, is a standard for development of enterprise applications based on reusable components in a multi-tier environment. In addition to the features of the Java 2 Platform, Standard Edition (J2SE) J2EE adds support for Enterprise Java Beans (EJBs), Java Server Pages (JSPs), Servlets, XML and many more. The J2EE architecture is used to bring together existing technologies and enterprise applications in a single, manageable environment.

Derby is a J2EE-conformant component in a distributed J2EE system. As such, Derby is one part of a larger system that includes, among other things, a JNDI server, a connection pool module, a transaction manager, a resource manager, and user applications. Within this system, Derby can serve as the resource manager.

For more information on J2EE, see the J2EE specification available at *http://java.sun.com/javaee/reference/* .

In order to qualify as a resource manager in a J2EE system, J2EE requires these basic areas of support:

- JNDI support

  Allows calling applications to register names for databases and access them through those names instead of through database connection URLs. Implementation of one of the JDBC interfaces, *javax.sql.DataSource,* provides this support.
- Connection pooling

  A mechanism by which a connection pool server keeps a set of open connections to a resource manager (Derby). A user requesting a connection can get one of the available connections from the pool. Such a connection pool is useful in client/server environments because establishing a connection is relatively expensive. In an embedded environment, connections are much cheaper, making the performance advantage of a connection pool negligible. Implementation of two of the JDBC interfaces, *javax.sql.ConnectionPoolDataSource and javax.sql.PooledConnection*, provide this support.
- XA support

  XA is one of several standards for distributed transaction management. It is based on two-phase commit. The *javax.sql.XAxxx* interfaces, along with *java.transaction.xa* package, are an abstract implementation of XA. For more information about XA, see *X/Open CAE Specification-Distributed Transaction Processing: The XA Specification*, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3. Implementation of the JTA API, the interfaces of the java.transaction.xa package (*javax.sql.XAConnection*,*javax.sql.XADataSource*,*javax.transaction.xa.XAResource*,*javax.transactio* and *javax.transaction.xa.XAException*), provides this support.

With the exception of the core JDBC interfaces, these interfaces are not visible to the end-user application; instead, they are used only by the other back-end components in the system.

Note: For information on the classes that implement these interfaces and how to use Derby as a resource manager, see Chapter 6, "Using Derby as a J2EE Resource Manager" in the *Java DB Developer's Guide*.

The JTA API

The JTA API is made up of the two interfaces and one exception that are part of the *java.transaction.xa* package. Derby fully implements this API.

- *javax.transaction.xa.XAResource*
- *javax.transaction.xa.Xid*
- *javax.transaction.xa.XAException*

Notes on Product Behavior

Recovered Global Transactions

Using the *XAResource.prepare* call causes a global transaction to enter a prepared state, which allows it to be persistent. Typically, the prepared state is just a transitional state before the transaction outcome is determined. However, if the system crashes, recovery puts transactions in the prepared state back into that state and awaits instructions from the transaction manager.

XAConnections, user names and passwords

If a user opens an *XAConnection* with a user name and password, the transaction it created cannot be attached to an *XAConnection* opened with a different user name and password. A transaction created with an *XAConnection* without a user name and password can be attached to any *XAConnection*.

However, the user name and password for recovered global transactions are lost; any *XAConnection* can commit or roll back that in-doubt transaction.
Note: Use the network client driver's XA DataSource interface (org.apache.derby.jdbc.ClientXADataSource) when XA support is required in a remote (client/server) environment.

javax.sql: JDBC Interfaces

This section documents the JDBC interfaces that Derby implements for J2EE compliance.

For more details about these interfaces, see the API documentation for your version of the Java Development Kit, which you can find at http://java.sun.com/javase/reference/api.jsp.

- *javax.sql.DataSource*

  An interface that is a factory for connections to the physical data source that the object represents. An object that implements the DataSource interface will typically be registered with a naming service based on the Java Naming and Directory (JNDI) API.
- *javax.sql.ConnectionPoolDataSource* and *javax.sql.PooledConnection*

  Establishing a connection to the database can be a relatively expensive operation in client/server environments. Establishing the connection once and then using the same connection for multiple requests can dramatically improve the performance of a database.

  The Derby implementation of `ConnectionPoolDataSource` and `PooledConnection` interfaces allows a connection pool server to maintain a set of

such connections to the resource manager (Derby). In an embedded environment, connections are much cheaper and connection pooling is not necessary.

- *javax.sql.XAConnection*

  An *XAConnection* produces an *XAResource*, and, over its lifetime, many *Connections*. This type of connection allows for distributed transactions.
- *javax.sql.XADataSource*

  An *XADataSource* is simply a *ConnectionPoolDataSource* that produces *XAConnections*.

In addition, Derby provides three methods for *XADataSource, DataSource, and ConnectionPoolDataSource*. Derby supports a number of additional data source properties:

- *setCreateDatabase(String create)*

  Sets a property to create a database at the next connection. The string argument must be "create".
- *setShutdownDatabase(String shutdown)*

  Sets a property to shut down a database. Shuts down the database at the next connection. The string argument must be "shutdown".

Note: Set these properties before getting the connection.

Derby API

Derby provides Javadoc HTML files of API classes and interfaces in the *javadoc* subdirectory.

This appendix provides a brief overview of the API. Derby does not provide the Javadoc for the *java.sql* packages, the main API for working with Derby, because it is included in the JDBC API. For information about Derby's implementation of JDBC, see JDBC reference.

This document divides the API classes and interfaces into several categories. The stand-alone tools and utilities are java classes that stand on their own and are invoked in a command window. The JDBC implementation classes are standard JDBC APIs, and are not invoked on the command-line. Instead, you invoke these only within a specified context within another application.

Stand-alone tools and utilities

These classes are in the packages *org.apache.derby.tools*.

- *org.apache.derby.tools.ij*

  An SQL scripting tool that can run as an embedded or a remote client/server application. See the *Java DB Tools and Utilities Guide*.
- *org.apache.derby.tools.sysinfo*

  A command-line, server-side utility that displays information about your JVM and Derby product. See the *Java DB Tools and Utilities Guide*.
- *org.apache.derby.tools.dblook*

  A utility to view all or parts of the Data Definition Language (DDL) for a given database. See the *Java DB Tools and Utilities Guide*.

JDBC implementation classes

JDBC driver

This is the JDBC driver for Derby:

- *org.apache.derby.jdbc.EmbeddedDriver*

  Used to boot the embedded built-in JDBC driver and the Derby system.
- *org.apache.derby.jdbc.ClientDriver*

  Used to connect to the Derby Network Server in client-server mode.

See the *Java DB Developer's Guide*.

Data Source Classes

These classes are all related to Derby's implementation of *javax.sql.DataSource* and related APIs. For more information, see the *Java DB Developer's Guide*.

Each of these classes has two variants. Use the first variant if your application runs on JDK 1.5 or lower. Use the second variant (the one ending in "40") if your application runs on JDK 1.6 or higher.

Embedded environment:

- *org.apache.derby.jdbc.EmbeddedDataSource* and *org.apache.derby.jdbc.EmbeddedDataSource40*
- *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource* and *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40*
- *org.apache.derby.jdbc.EmbeddedXADataSource* and *org.apache.derby.jdbc.EmbeddedXADataSource40*

Client-server environment

- *org.apache.derby.jdbc.ClientDataSource* and *org.apache.derby.jdbc.ClientDataSource40*
- *org.apache.derby.jdbc.ClientConnectionPoolDataSource* and *org.apache.derby.jdbc.ClientConnectionPoolDataSource40*
- *org.apache.derby.jdbc.ClientXADataSource* and *org.apache.derby.jdbc.ClientXADataSource40*

Miscellaneous utilities and interfaces

- *org.apache.derby.authentication.UserAuthenticator*
- An interface provided by Derby. Classes that provide an alternate user authentication scheme must implement this interface. For information about users, see "Working with User Authentication" in Chapter 7 of the *Java DB Developer's Guide*.

Supported territories

The following is a list of supported territories:

Territory	Derby territory setting (derby.territory)
Chinese (Simplified)	zh_CN
Chinese (Traditional)	zh_TW
Czech	cs
French	fr
German	de_DE
Hungarian	hu
Italian	it
Japanese	ja_JP
Korean	ko_KR
Polish	pl
Portuguese (Brazilian)	pt_BR
Russian	ru
Spanish	es

Derby limitations

The section lists the limitations associated with Derby.

Limitations for database manager values

Table 104. Database manager limitations
The following table lists limitations on various Database Manager values in Derby.

Value	Limit
Maximum columns in a table	1,012
Maximum columns in a view	5,000
Maximum number of parameters in a stored procedure	90
Maximum indexes on a table	32,767 or storage capacity
Maximum tables referenced in an SQL statement or a view	storage capacity
Maximum elements in a select list	1,012
Maximum predicates in a WHERE or HAVING clause	storage capacity
Maximum number of columns in a GROUP BY clause	32,677
Maximum number of columns in an ORDER BY clause	1,012
Maximum number of prepared statements	storage capacity
Maximum declared cursors in a program	storage capacity
Maximum number of cursors opened at one time	storage capacity
Maximum number of constraints on a table	storage capacity
Maximum level of subquery nesting	storage capacity
Maximum number of subqueries in a single statement	storage capacity
Maximum number of rows changed in a unit of work	storage capacity
Maximum constants in a statement	storage capacity
Maximum depth of cascaded triggers	16

DATE, TIME, and TIMESTAMP limitations

The following table lists limitations on date, time, and timestamp values in Derby.
Table 105. DATE, TIME, and TIMESTAMP limitations

Value	Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-23.59.59.999999

Limitations on identifier length

Table 106. Identifier length limitations
The following table lists limitations on identifier lengths in Derby.

Identifier	Maximum number of characters allowed
constraint name	128
correlation name	128
cursor name	128
data source column name	128
data source index name	128
data source name	128
savepoint name	128
schema name	128
unqualified column name	128
unqualified function name	128
unqualified index name	128
unqualified procedure name	128
parameter name	128
unqualified trigger name	128
unqualified table name, view name, stored procedure name	128

Numeric limitations

There are limitations on the numeric values in Derby.

Table 107. Numeric limitations

Value	Limit
Smallest INTEGER	-2,147,483,648
Largest INTEGER	2,147,483,647

Value	Limit
Smallest BIGINT	-9,223,372,036,854,775,808
Largest BIGINT	9,223,372,036,854,775,807
Smallest SMALLINT	-32,768
Largest SMALLINT	32,767
Largest decimal precision	31
Smallest DOUBLE	-1.79769E+308
Largest DOUBLE	1.79769E+308
Smallest positive DOUBLE	2.225E-307
Largest negative DOUBLE	-2.225E-307
Smallest REAL	-3.402E+38
Largest REAL	3.402E+38
Smallest positive REAL	1.175E-37
Largest negative REAL	-1.175E-37

String limitations

Table 108. String limitations
The following table contains limitations on string values in Derby.

Value	Maximum Limit
Length of CHAR	254 characters
Length of VARCHAR	32,672 characters
Length of LONG VARCHAR	32,700 characters
Length of CLOB	2,147,483,647 characters
Length of BLOB	2,147,483,647 characters
Length of character constant	32,672
Length of concatenated character string	2,147,483,647
Length of concatenated binary string	2,147,483,647
Number of hex constant digits	16,336
Length of DOUBLE value constant	30 characters

XML limitations

The following table lists the limitations on XML data types in Derby.

Table 109. XML limitations

Issue	Limitation
Length of XML	2,147,483,647 characters

Issue	Limitation
Use of XML operators	Requires that the JAXP parser classes (such as Apache Xerces) and the Apache Xalan classes are in the classpath. Attempts to use XML operators without these classes in the classpath result in an error. In some situations, you may need to take steps to place the parser and Xalan in your classpath. See "XML data types and operators" in the *Java DB Developer's Guide* for details.

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.